ITALIAN INSTITUTE OF TECHNOLOGY
&
UNIVERSITY OF GENOVA

PHD PROGRAM IN BIOENGINEERING AND ROBOTICS

Exploiting Prior Knowledge in Robot Motion Skills Learning

by

Brian Delhaisse
(鍋島ブライアン)

Thesis submitted for the degree of *Doctor of Philosophy* (31° cycle)

December 2019

Prof. Darwin G. Caldwell	Advisor
Dr. Leonel Rozo	Supervisor
Prof. Giorgio Cannata	Head of the PhD program

Thesis Jury:

Dr. Luka Peternel, *Delft University of Technology (TU Delft)*	External examiner
Prof. Dongheui Lee, *Technical University of Munich (TUM)*	External examiner

Dibris

Department of Informatics, Bioengineering, Robotics and Systems Engineering

I would like to dedicate this thesis to my parents, family, and friends for their love and unconditional support throughout the years.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 65,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 150 figures.

Brian Delhaisse
February 2020

Acknowledgements

My PhD journey actually began during the final year of my master's degree, when my then-supervisor Mauro Birattari suggested me to pursue a PhD degree. For that, I am grateful to him as I do think pursuing a PhD was definitely the right choice for me. When I arrived to the Istituto Italiano di Tecnologia (IIT) in November 2015, my supposed-to-be supervisor left for another position, and 3 Postdocs (Arash, Przemek, and Jinoh) volunteered to supervise me. While our interests didn't really align with one another, I am really grateful to them. I am thankful to Arash for explaining me and providing me with references in robotics. Coming more from a computer science degree, it unquestionably helped me to fill my lack of knowledge in that field. I am also grateful to Przemek for the various conversations we had about robot locomotion from which I also learned a lot. While he left atfer one year of my PhD, I definitely enjoyed our conversations about different topics.

Most importantly, I would like to thank Leonel Rozo for proposing to be my supervisor in the field I have always been interested in, namely, robot learning. His support, guidance, and advices helped me greatly throughout my PhD. Not only that, he provided me an environment where I could explore and work freely on different ideas. For that, I am grateful as it allowed me to learn so much during my PhD. I would also like to thanks my advisor Darwin Caldwell for the opportunity to perform research at IIT, but also his support to my request for extending one extra year my PhD.

As the last member of the learning group, I would like to acknowledge and thank as well its past members: Leonel, Joao, Domingo, Martijn, Yanlong, and Fares. Their knowledge about different subfields in robot learning made it very interesting to be part of this group. I am also thankful for their valuable input during our group meetings, and our trips to various restaurants in Genova. I would like to especially thank my friend Joao for our various interesting conversations about different topics ranging from politics to philosophy.

I am also thankful to the various PhD friends that I made at IIT. I would like to thank Songyan, Zeyu, Rajesh, Yangwei, Anh, Malgorzata, and Vishnu for our various conversations, and the activities that we did together. They definitely made this PhD more joyful. I would like to thank as well all the members from ADVR, past and present, from which I have

learned so much during my PhD. They also played a key role in my personal and academic growth. I am also grateful to the ADVR secretaries, especially Silvia and Floriana, for handling all administration related to the travel expenses for the various conferences that I attended to, and their support for my PhD extension. I would also like to thank Dr. Peternel, and Prof. Dr. Lee for accepting to review my thesis, and their valuable feedback.

Finally, I would like to thank my family and friends. My parents for their unconditional love and support, as well as providing me with an environment where I could pursue anything I was curious about since a very young age. My brothers and cousins for their love and support. My closest friends Melissa, Alexis, Daniele, and Ségolène; you are the best friends I could hope for! I would especially like to thank my then-girlfriend Melissa who came to live with me in Italy for the first year of my PhD. While it didn't work out at the end between us, I am indebted to you, and you will always remain one of my closest friends. Lastly, I would like to thank my late grandmother and godmother for their past guidance; you will always remain in my heart.

Merci pour votre soutien! 信じてくれて、ありがとうございます。

Abstract

This thesis presents a new robot learning framework, its application to exploit prior knowledge by encoding movement primitives in the form of a motion library, and the transfer of such knowledge to other robotic platforms in the form of shared latent spaces. In this abstract, we present the motivation and objectives, the developed framework and methods, as well as our contributions to the robot learning field.

Objectives

In robot learning, it is often desirable to have robots that learn and acquire new skills rapidly. However, existing methods are specific to a certain task defined by the user, as well as time consuming to train. This includes for instance end-to-end models that can require a substantial amount of time to learn a certain skill. Such methods often start with no prior knowledge or little, and move slowly from erratic movements to the specific required motion. This is very different from how animals and humans learn motion skills. For instance, zebras in the African Savannah can learn to walk in few minutes just after being born. This suggests that some kind of prior knowledge is encoded into them. Leveraging this information may help improve and accelerate the learning and generation of new skills. These observations raise questions such as: how would this prior knowledge be represented? And how much would it help the learning process? Additionally, once learned, these models often do not transfer well to other robotic platforms requiring to teach to each other robot the same skills. This significantly increases the total training time and render the demonstration phase a tedious process. Would it be possible instead to exploit this prior knowledge to accelerate the learning process of new skills by transferring it to other robots? These are some of the questions that we are interested to investigate in this thesis. However, before examining these questions, a pratical tool that allows one to easily test ideas in robot learning is needed. This tool would have to be easy-to-use, intuitive, generic, modular, and would need to let the user easily implement different ideas and compare different models/algorithms. Once implemented, we would then be able to focus on our original questions.

Framework and Methods

In research, it is often necessary to quickly test ideas and compare different methods proposed in the literature. This requires the need for a generic, flexible, and modular framework. Modularity encourages the implementation and reuse of different components, while genericity enables the framework to be general enough to be used in different scenarios. Finally, flexibility allows to combine different components easily and do not constraint the user on a specific way of coding. We provide a robot learning framework, namely *PyRoboLearn*, that provides and combines different learning paradigms, including imitation and reinforcement learning. This framework already has more than 100,000 lines of Python code and include various robots, environments, learning models, algorithms, controllers and others. Additionally, it enables one to test a policy in simulation as well as to transfer it on a real robot through the ROS middleware.

Once such framework is available, we can focus our attention on the questions raised above, notably, the representation of prior knowledge for the generation of new skills. Inspired by biological systems, such as vertebrates and invertebrates, that use a finite number of movement primitives and superposed these to represent motions, we formulated a dynamic motion library. This data-driven library is built using dynamic movement primitives and spectral decomposition. It allows to represent motions in a compact way in terms of what we refer as eigenforces. These primitives as their biological counterparts obey the superposition principle, and any motions can be expressed as a linear combination of these primitives. We investigate its use on movement recognition and generation, as well as its adaptability on a reinforcement learning problem. Finally, we provide a preliminary work on how to couple such library with a perceptual system. This last system is represented as a convolutional neural network and combined with our library to generate trajectories.

Once a library has been built, it becomes interesting to see how such prior knowledge can be transferred to other robots which have slightly different kinematic structures. By realizing that most movements lie on lower dimensional submanifold as demonstrated by our motion library, we investigated how such latent space can be shared and transferred to other robotic platforms. This ultimately led us to use a non-linear, bayesian, and non-parametric model namely a shared Gaussian process latent variable model. By applying this model on our problem, we demonstrated that we could transfer knowledge between different robots, leading to an acceleration in the learning process of new skills for other robots.

Contributions

Our contribution can be divided into two categories; a practical contribution through the

implementation of a new robot learning framework[1], and two theoretical contributions, namely, the building of a dynamic motion library to encode movement primitives and its applications, as well as the transfer of a shared latent space to accelerate the learning of skills on other robotic platforms. Both last contributions addresses the representation of prior knowledge and how it can be exploited to accelerate the learning process when acquiring new skills. We review briefly each contribution in the following paragraphes.

Our first contribution addresses the robot learning framework. We provide an easy-to-use, generic and modular framework to test different ideas in robotics and machine learning. Compared to previous frameworks, it provides a panoply of functionalities, models and algorithms, as well as different learning paradigms such as imitation and reinforcement learning into one single framework. Futhermore, it is agnostic to the considered simulator and allows to easily transfer a policy to the real robot through a middleware layer.

Our second contribution deals with the encoding of movement primitives, and the construction of a dynamic motion library. This library allows to compactly encode and store motion primitives by exploiting the superposition principle. By adapting it to the considered task we can decouple some degrees of freedom, and learn faster certain skills.

Our third and final contribution covers the use and transfer of shared latent spaces between similar robotic kinematic structures to speed up the learning of new kinematic skills by these other robots.

[1] implemented in Python.

Notation and Symbols

This thesis mainly follows the notation guidelines described in the following table.

Notation	Description	Meaning
a	normal case	scalar value
$\mathbb{A}$	upper-case with style font	Set
$\boldsymbol{a}$	bold lower-case	vector
$\boldsymbol{A}$	bold upper-case	matrix
$\boldsymbol{A}^\top$	superscript script-style T	matrix transpose
$\boldsymbol{A}^\dagger$	superscript dagger	pseudo-inverse of a matrix
$\boldsymbol{a}_i$	subscript letter	index indicator

The following is a list of commonly used symbols throughout the thesis.

Notation	Meaning
$\boldsymbol{q}, \dot{\boldsymbol{q}}, \ddot{\boldsymbol{q}}$	joint positions, velocities, and accelerations respectively
$\boldsymbol{\tau}$	joint torques
$\boldsymbol{x}_w, \dot{\boldsymbol{x}}_w, \ddot{\boldsymbol{x}}_w$	Cartesian positions, velocities, and accelerations with respect to world frame w
N	number of data points
D	dimensionality of a data point
T	length of a trajectory
K	number of hyperparameters
θ	parameters of a model
Φ	hyper-parameters of a model
$\psi(.)$	basis function
$\boldsymbol{I}$	identity matrix
$\boldsymbol{J}$	Jacobian matrix
$\boldsymbol{H}(\boldsymbol{q})$	Inertia matrix
π	mathematical constant pi
$\mathbb{R}$	the set of real numbers
$\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$	multivariate normal distribution with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$
${}_i\boldsymbol{R}_k^j$	rotation matrix from frame j to frame k expressed in frame i
$\mathcal{O}(.)$	computational time or space complexity (specified in the text which type it is)
$p(.)$	probability density function
$\mathbb{T}$	space of all trajectories

Table of contents

List of figures

List of tables

Chapter 1

Introduction

1.1 Background

Robots are expected to be part of our daily lives performing diverse tasks in unstructured environments. To enable these robots to help us and carry out these tasks, two main schools of thoughts have emerged. The first one consists to model mathematically as much as possible the dynamics of the robot and its interaction with the environment, and from that, to manually design controllers to deal with the considered task. This can be categorized as the *classical robotics/engineering approach*. The second one instead consists to exploit the abundance of data around us, and extract useful information from it using statistics, probabilities, and machine learning, and learn a policy[1] to perform the task. This is often referred as the *robot learning approach*. To understand the difference between these two views, consider the task of riding a bike. One could mathematically model the whole dynamics of the system to a certain accuracy, and design a controller to perform this specific task. However, this often results in designed controllers that deal with specific scenarios of the task. Additionally, these methods make several assumptions and simplification when modeling the robot and its interaction with the environment (such as linearization of the dynamics, selection of a certain friction model, and others). Another way would be to let the agent try by itself to ride a bike and learn from its failures and successes on how to ride it properly. The last approach is mostly similar to how human beings operate. The second approach has the benefits of being more generic and widely more applicable to different scenarios because of the available amount of data. Learning models, such as policies, however can be time-consuming to

[1] In this thesis, we will refer a policy as a learnable controller which optimizes its parameters or hyperparameters based on some collected data, and refer simply a controller as a manually designed one (where no data is involved)

train requiring a huge number of samples to extract the useful pieces of information from. End-to-end models for instance require millions even billions of data points to learn a certain behavior. Such methods often start with no prior knowledge or little, and move slowly from erratic movements to the specific required motion. In contrast, animals and humans, despite their high number of degree of freedoms, display elaborated movements, learn and adapt quickly when facing unseen situations. For instance, animals in the African Savannah like zebras learn quickly how to walk and run as soon after they are born in order to escape predators. This suggests that some sort of prior knowledge is encoded into them [10, 28, 64]. Leveraging this information to robot learning [5, 40, 41, 18] may help improve and accelerate the learning process and generation of new skills. Considering multiple robots of similar morphologies, transferring such prior knowledge between these might also accelerate the learning process making it easier for the user to teach new skills to other robots as well.

1.2 Proposed Approaches and Contributions

Our contributions can be subdivided into two categories: practical and theoretical contributions. In each of the following subsection, we describe our contribution as well as a general overview of the proposed framework and approaches. We start by our pratical contribution, namely the robot learning framework, followed by our two theoretical contributions. These last two include the building of a dynamic motion library as well as the exploitation of shared latent space for transfer learning.

1.2.1 Robot Learning Framework

Before being able to test ideas and compare methods in robot learning, a framework is much needed. Such framework needs to be easy-to-use, modular, generic, and flexible. Modularity encourages the implementation and reuse of different components, while genericity enables the framework to be general enough to be used in different scenarios. Finally, flexibility allows to combine different components easily and do not constraint the user on a specific way of coding. We provide a robot learning framework that provides and combines different learning paradigms, including imitation and reinforcement learning. This framework already has around 100,000 lines of Python code and include various robots, environments, learning models, algorithms, controllers, simulators, and others. Additionally, it enables one to test a policy in simulation as well as to transfer it on a real robot through the ROS middleware.

A general overview of our proposed framework is depicted in Fig. 1.1, and a more detailed description is provided in Chapter 2.

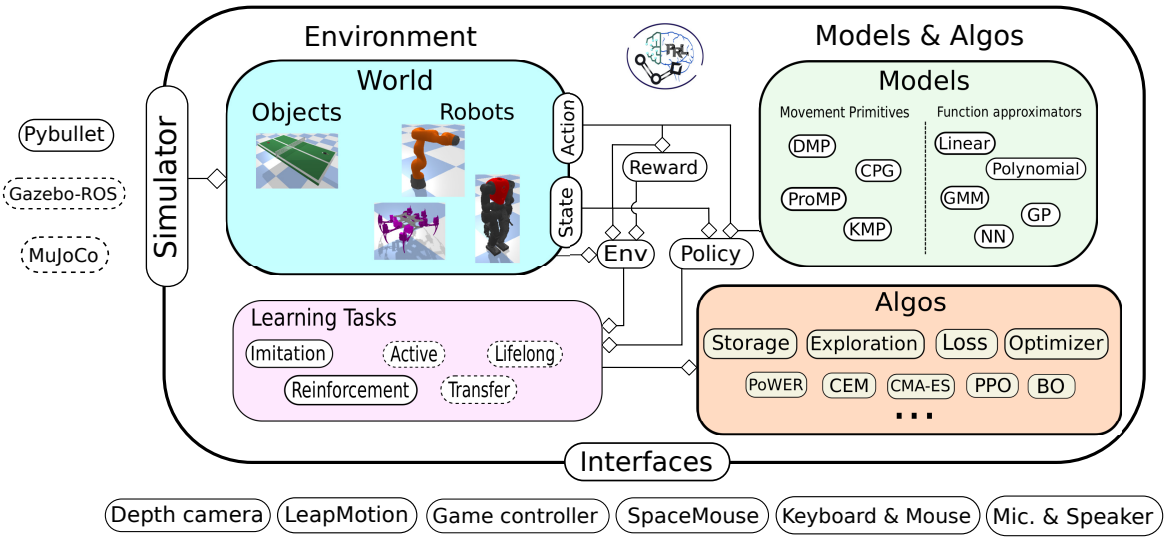

Figure 1.1 Overview of the *PyRoboLearn* (PRL) architecture where we abstract each robot learning concept, adopt a modular programming approach, minimize the modules coupling, and favor composition over inheritance [30] to increase the flexibility. PRL functionalities cover seven main axes: simulators, worlds, robots, learning paradigms, interfaces, learning models, and learning algorithms.

1.2.2 Dynamic Motion Library

Incorporating prior knowledge is primordial to ease and fasten the learning of robot skills. Inspired by how biological systems, such as vertebrates and invertebrates, learn to move using movement primitives, we built a dynamic motion library that contains movement primitives and exploits the superposition principle. Exploiting this principle decreases substantially the number of independent primitives needed, and enables to represent any other primitives by a weighted sums of these independent ones. For this purpose, we made use of dynamic movement primitives and the spectral theorem to generate our library. A general overview of our proposed framework is shown in Fig. 1.2, and is investigated in Chapter 3.

1.2.3 Transfer Learning of Shared Latent Spaces

Once a robot has learned a certain skill, it becomes interesting to study how this knowledge can be transfered to another robot with a different kinematic structure. This is notably useful in industrial robotics where instead of having an operator spending time re-teaching all the

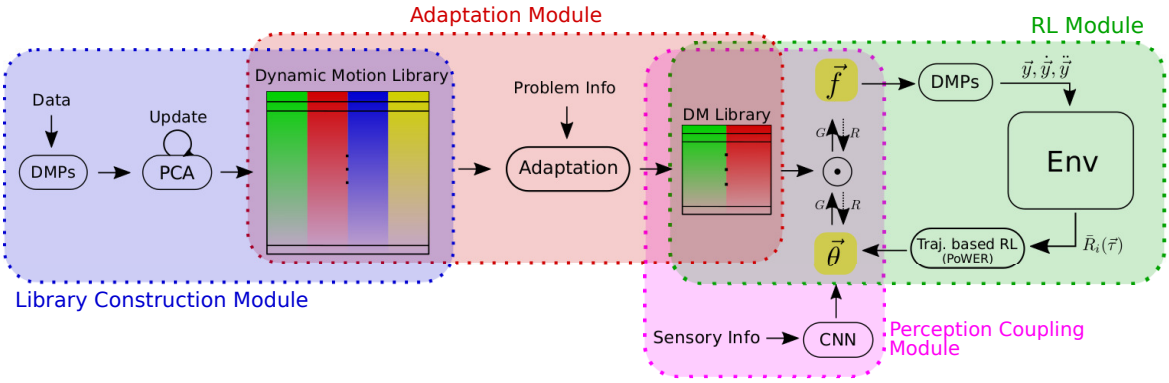

Figure 1.2 Overview of the proposed motion library approach subdivided into 4 modules: a *library construction module*, an *adaptation module*, a *reinforcement learning (RL) module*, and a *perception coupling module*. These modules will be described in more details in Chapter 3.

skills shown on a previous robotic platform, the acquired knowledge is transfered to the other robots to ease and fasten the learning process of new skills. Using the dynamic motion library, we learned that movements can be projected on a lower dimensional submanifold. Here, we investigated the use of shared latent spaces for robot skills and how they can be transfered from one robotic platform to another. For this purpose, we used a non-linear, bayesian, and non-parametric dimensionality reduction model namely a shared Gaussian process latent variable model (shared GP-LVM), and transfered the joint latent space learned between one robot and a human operator to other robots requiring only to re-train the mapping between the latent space to the other robot space. A general overview of the proposed methodology is depicted in Fig. 1.3, and a deeper insight is provided in Chapter 4.

1.3 Thesis Outline

The thesis is centered around the concept of exploiting prior knowledge to enable robots to learn faster. For this purpose, it is splitted into 3 main parts: a robot learning framework, a dynamic motion library, and shared latent spaces for transfer learning. The first part is our practical contribution while the two others are our theoretical contributions to the robot learning field. A graphical representation of the thesis outline is provided in Fig. 1.4.

The thesis is structured such that each chapter can be read independently. As such, in each chapter, we introduce the necessary background and review the corresponding state of the art before diving into our proposed approach and contribution.

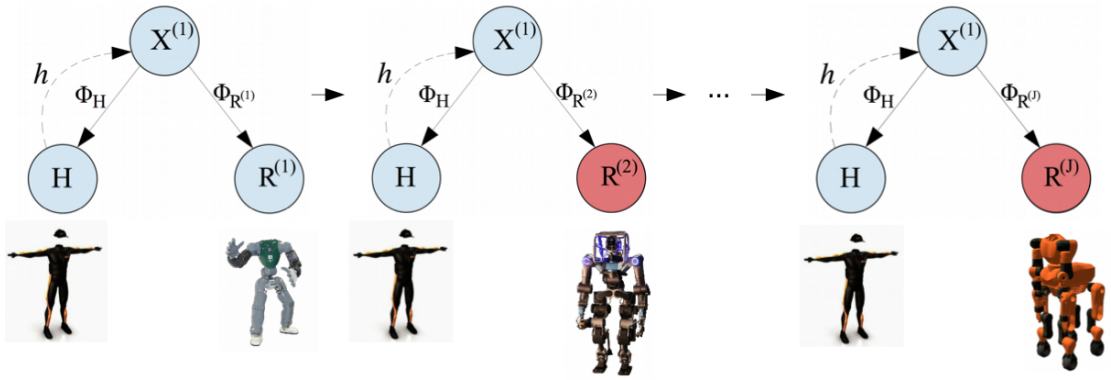

Figure 1.3 A shared GP-LVM fully trained on one robot $\boldsymbol{R}^{(1)}$, that is, the hyperparameters $\boldsymbol{\Phi}_H$, $\boldsymbol{\Phi}_{R^{(1)}}$, and the latent coordinates $\boldsymbol{X}^{(1)}$ are jointly optimized. This model is then transferred to another robot $\boldsymbol{R}^{(2)}$ in which the latent coordinates and the hyperparameters $\boldsymbol{\Phi}_H$ are maintained fixed while the hyperparameters $\boldsymbol{\Phi}_{R^{(2)}}$ for the new latent-to-output mapping are optimized. This process is carried out over all the other robots $\boldsymbol{R}^{(j)}$, $\forall j \in \{3,...,J\}$. Once the optimization process is over, new human input data are given to the system which produce the corresponding output data for each robot.

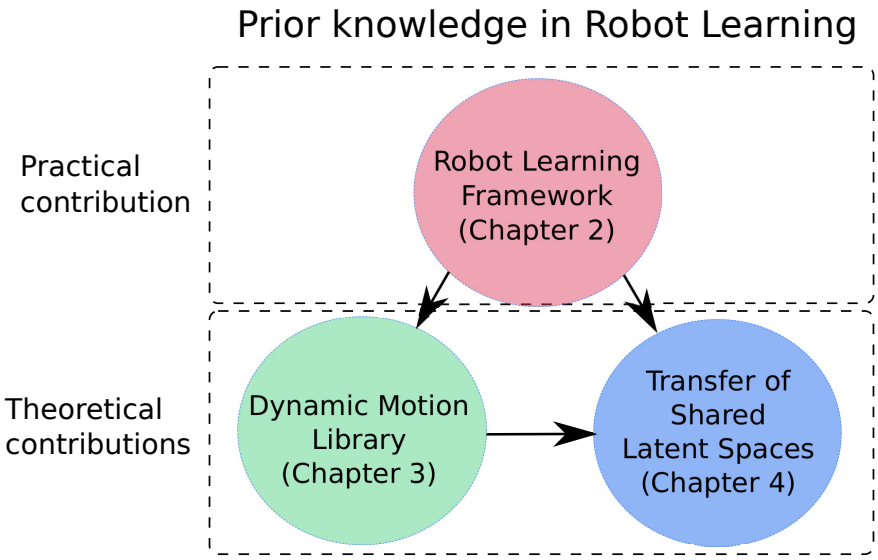

Figure 1.4 Graphical representation of the thesis content.

1.4 Supplementary Material

The framework, models and methods used in this work have been implemented in Python. The framework is named `pyrobolearn` and can be found online on Github at https://robotlearn.github.io/pyrobolearn/, and is currently licensed under the GPLv3 license. The repository

contains several examples, tutorials, and documentation. Videos as well as supplementary materials associated with each chapter can also be found at the following links:

- Chapter 2: https://robotlearn.github.io/pyrobolearn/
- Chapter 3: https://sites.google.com/view/dynamic-motion-library/home
- Chapter 4: https://gitlab.com/bdelhaisse/HSGPLVM

Chapter 2

A Robot Learning Framework

As mentioned in the Introduction section, a framework is required to test different ideas and compare different methods. On the quest for building autonomous robots, several robot learning frameworks with different functionalities have recently been developed. Yet, frameworks that combine diverse learning paradigms (such as imitation and reinforcement learning) into a common place are scarce. Existing ones tend to be robot-specific, and often require time-consuming work to be used with other robots. Also, their architecture is often weakly structured, mainly because of a lack of modularity and flexibility. This leads users to reimplement several pieces of code to integrate them into their own experimental or benchmarking work. To overcome these issues, we introduce *PyRoboLearn*, a new Python robot learning framework that combines different learning paradigms into a single framework. Our framework provides a plethora of robotic environments, learning models and algorithms. *PyRoboLearn* is developed with a particular focus on modularity, flexibility, generality, and simplicity to favor (re)usability. This is achieved by abstracting each key concept, undertaking a modular programming approach, minimizing the coupling among the different modules, and favoring composition over inheritance for better flexibility. We demonstrate the different features and utility of our framework through different use cases.

2.1 Introduction

Recent advances in machine learning for robotics have produced several (free and) open-source libraries and frameworks. These ease the understanding of new concepts, allow for the comparison of different methods, provide testbeds and benchmarks, promote reproducible research, and enable the reuse of existing software. Nevertheless, several frameworks suffer from a lack of flexibility and generality due to poor design choices. Lack of abstraction and

modularity with high dependency among modules hinder code reuse. This problem worsens when the user needs to combine different incompatible codes together, or to integrate an existing one into her own code. Some frameworks force to follow a standard, which might not suit the user needs. However, bypassing code standards is not a good coding practice as many useful functionalities might be missed. Complying to their standard requires to modify the original code, interface (possibly) incompatible frameworks, and/or reimplement parts of the framework. This creates unnecessary overheads that considerably affect the research activities, leaving less time to create modular and flexible code, and therefore ad-hoc code that is hardly reusable is produced.

Available frameworks in *robot learning* [87] can be classified into two categories: "simulated environments" [12, 124, 45, 113, 16, 24, 37, 4, 21] and "models and algorithms" [84, 31, 50, 58, 22, 88]. In both, frameworks tend to focus on specific learning paradigms such as imitation learning (IL) [7] or reinforcement learning (RL) [111], and do not exploit their shared features, such as an environment, trainable policies, states/actions, and loss functions. In IL, a teacher provides demonstration data while for RL a reward signal is returned by the environment, which results in different training algorithms. The majority of frameworks that provide simulated environments focus either on RL [12, 124, 45, 113, 16, 24], or to a less extent on IL [37, 4, 21], which limits their applicability. As IL and RL differ on few aspects, their integration and design into a single learning framework provides interesting opportunities. For example, IL can be used to initialize a policy which is then fine-tuned using RL, leading to safer and faster policy search [18]. However, current environment frameworks rarely exploit this feature.

To better illustrate our point, let us consider an RL setting where an environment inherits from an OpenAI Gym environment [12], which several frameworks use [124, 45, 16, 24]. Such environment includes the definition of state-action spaces, environment, and reward function. Also, let us consider an environment that includes an inverted pendulum on a cart. The state consists of the cart position and velocity, and the angular position and velocity of the pole. A simple reward function may count the number of time steps the cart could balance the pole. Finally, let us define a neural network policy that is specified outside the environment, which takes the $4D$ state vector and outputs the action. Now, assume that the user wants to test the performance of a new model/algorithm on a double inverted pendulum on a cart. In this case, the user would have to define manually a new environment with a new robot, and a larger dimensional state vector. This, in turn, affects the policy representation. Moreover, if the user wishes to experiment different reward functions, she would have to change them directly in the environment definition.

The above procedure is not efficient and does not scale. A better approach is to have the state to change its dimensionality automatically as the robot varies, and the neural network policy architecture to adapt accordingly. The reward function could be defined outside the environment and then provided to it. This lack of simplicity, modularity and flexibility along with the lack of a common framework regrouping different learning paradigms is what motivated us to create *PyRoboLearn*. For this purpose, we adopted a modular and SOLID programming approach [65], abstract important concepts, minimized the dependencies between modules, and favored composition over inheritance to increase flexibility. *PyRoboLearn* provides diverse environments, learning models and algorithms, and permits to easily and quickly experiment ideas by combining diverse features and modules.

2.2 Related Work

To reach high usability, our framework is written in Python and uses the PyTorch library [84] as backend. Frameworks in other languages are often prone to errors and not beginner-friendly. As such, we do not review the literature of frameworks written in other languages. In general, robot learning frameworks can be mainly categorized as: environment-based and model-based. We start by reviewing the literature of environment-based frameworks.

In IL, few environments have been proposed, notably SMILE [37] and the Freiberg Robot Simulator [4][1], but both focus on specific robotic platforms and use different programming languages. In contrast, multiple environments have been proposed for RL. One of the most used frameworks is OpenAI Gym [12] from which other frameworks have derived. OpenAI Gym provides environments in games, control, and robotics. Each one inherits from the abstract Gym environment class, and defines the world, the agents, the states-actions, and the reward function inside its class. Inheritance is used over composition which limits flexibility as a new environment has to be created for each combination of worlds (including the agents), states and rewards. OpenAI Gym and the DeepMind control suite [113], use MuJoCo [115]. Since MuJoCo requires a license, Zamora et al. [124] extended the Gym framework with Gazebo and ROS. OpenAI later released roboschool [45], a free robotic framework to test RL algorithms. Built on the PyBullet [16] simulator, PyBullet-gym [24] was recently released. All these frameworks focus on RL and most inherit from the OpenAI gym, following the same protocol.

Few other frameworks such as Carla [21] and Airsim [102] support both IL and RL, but are designed for autonomous vehicles. Another new framework closely related to ours

[1] We tried to find this simulator online unsuccessfully.

is Surreal [25] which also supports IL and RL, but focuses only on manipulation tasks using the Baxter and Sawyer robots in MuJoCo. Other frameworks include the Gibson Environment [121] which focuses on perception learning and sim-to-real policy transfer, and the S-RL toolbox [91] which focuses on state representation learning. Both are out of the scope of the covered learning paradigms in this chapter. Recently, two new Python robot frameworks have been introduced: PyRobot [72] and PyRep [42]. The former provides a lightweight interface built on top of Gazebo-ROS [49, 90] with a focus on robotic manipulation and navigation, while the latter provides a Python wrapper around the V-REP simulator [95]. As our framework, they aim to be beginner-friendly but are mainly focused on the robotic application instead of being a complete robot learning framework. They can be better compared to a simulator such as PyBullet or MuJoCo. A table summarizing parts of the different characteristics of current robot learning frameworks that provide environments is depicted in Table 2.1.

Name	**OS**	**Python**	**Simulator**	**Paradigm**	**Robot**	**Problem**
Open-AI Gym [12]	OSX, Linux	2.7, 3.5	MuJoCo	RL	3D chars	Manip., Loco.
Gym-Gazebo [124]	Ubuntu 18.04	3	Gazebo + ROS	RL	<5 robots	Manip., Nav.
DeepMind Control Suite [113]	Ubuntu 14.04/16.04	2.7, 3.5	MuJoCo	RL	3D chars	Loco., Control
Roboschool [45]	OSX, Ubuntu/Debian	3	Bullet	RL	3D chars	Loco., Control
Pybullet Gym [16, 24]	OSX, Linux, Windows	2.7, 3.5	PyBullet	RL	3d chars, Atlas	Manip., Loco., Control
GibsonEnv [121]	Ubuntu	3.5	Bullet	PL/RL	3D chars, 5 robots	Perception, Nav.
Airsim [102]	Linux, Windows	3.5+	Unreal Engine/Unity	IL/RL	AV	Nav.
Carla [21]	Ubuntu 16.04+, Windows	2.7, 3.5	Unreal Engine	IL/RL	AV	Nav.
Surreal Robotics Suite [25]	OSX, Linux	3.5, 3.7	MuJoCo	IL/RL	Baxter, Sawyer	Manip.
S-RL Toolbox [91]	N/S	3.5+	PyBullet	RL/SRL	Kuka, OmniRobot	Manip., Nav.
PyRoboLearn	Ubuntu 16.04/18.04[2]	2.7, 3.5, 3.6	Agnostic (PyBullet)	IL/RL	60+	Manip., Loco., Control

Table 2.1 Comparisons between different robot learning frameworks that provide environments. PL stands for perception learning, SRL for state representation learning, AV for autonomous vehicles, Manip. for manipulation, Loc. for locomotion, and Nav. for navigation. Note that MuJoCo [115] is not open-source, requires a license, and depending on that last one might not be free. Also note that while the support for Python 2.7 will end in 2020, some simulators such as Gazebo-ROS and some libraries are still dependent on Python 2.7.

We now turn our attention to frameworks that provide models and algorithms. Several libraries have been proposed such as Sklearn [85], TensorFlow [1], PyTorch [84], GPy-

Torch [31], among others. As they use different backends (e.g. Numpy, TensorFlow or PyTorch), the models defined in one cannot use algorithms of the others. In our framework, we provide a common interface to existing models, and reimplement models that were not compatible. In RL, Garage (previously known as rllab) [22], baselines [36], and RLlib [58] are three popular libraries that provide out-of-the-box RL algorithms. The first two are coded in TensorFlow, while the latter is built on PyTorch. As for the environments, these model-based frameworks define their own standard which do not fit our modular framework. The main reasons being that learning algorithms are dependent of low-level concepts such as the environment and policies (i.e. models) making a possible integration harder. The next Table 2.2 summarizes the different aspects of frameworks that provides models and algorithms.

Name	OS	Python	Backend	Flexible
rllab/garage [22]	Linux, OSX	3.5+	TensorFlow	No
rllib [58]	Ubuntu 1[4,6,8], OSX 10.1[1-4]	2, 3	TensorFlow, PyTorch	No
stable-baselines [20, 36]	Ubuntu, OSX, Windows	3.5	TensorFlow	No
PyRoboLearn	Ubuntu 16.04/18.04	2.7, 3.5, 3.6	PyTorch	Yes

Table 2.2 Comparisons between different frameworks that provide reinforcement learning models and algorithms. Note that all these frameworks (except ours) focus on deep neural networks as their main models, and do not take into account other models such as movement primitives. Note that existing frameworks mostly focus on the reinforcement learning paradigm, and not on other paradigms such as imitation learning, active learning, transfer learning, and others.

2.3 Proposed Framework

PyRoboLearn (PRL) is designed to maximize modularity, flexibility, simplicity, and generality. Our first choice was the programming language. We chose Python[3] because of its simplicity to prototype new ideas, a fast learning curve, a huge amount of available libraries, and the ability to interact with the code. We also used PyTorch and Numpy for our learning models and algorithms. PyTorch has been selected because of its Pythonic nature, modularity and popularity in research.

[3]PyRoboLearn works in Python 2.7, 3.5, and 3.6, and has been tested on Ubuntu 16.04 and 18.04. While the support for Python 2.7 will end in 2020, many libraries used in robotics still depend on it.

Regarding the PyRoboLearn architecture, we abstracted each robot learning concept, adopted a modular programming approach, minimized the modules coupling, and favored composition over inheritance [30] to increase the flexibility [110]. Abstraction aims at identifying and abstracting different concepts into objects, and building high-level concepts on top of low-level ones. Modularity separates these concepts into independent and interchangeable building blocks that represent or implement a particular functionality. Composition combines different modules and thus different functionalities into a single one. Coupling measures how different modules depend on each other. A high coupling between two modules means they cannot work in a stand-alone fashion, while a low coupling means they depend on abstractions instead of concretions [65]. The aforementioned notions increased the framework flexibility while facilitating the reuse and integration of the various modules.

PRL functionalities cover eight main axes: simulators (with a possible middleware interface), worlds, robots, controllers, learning paradigms (including the definition of environments and states/actions), interfaces, learning models, and learning algorithms. Each of these components is described next. An overview of the framework is depicted in Fig. 2.1.

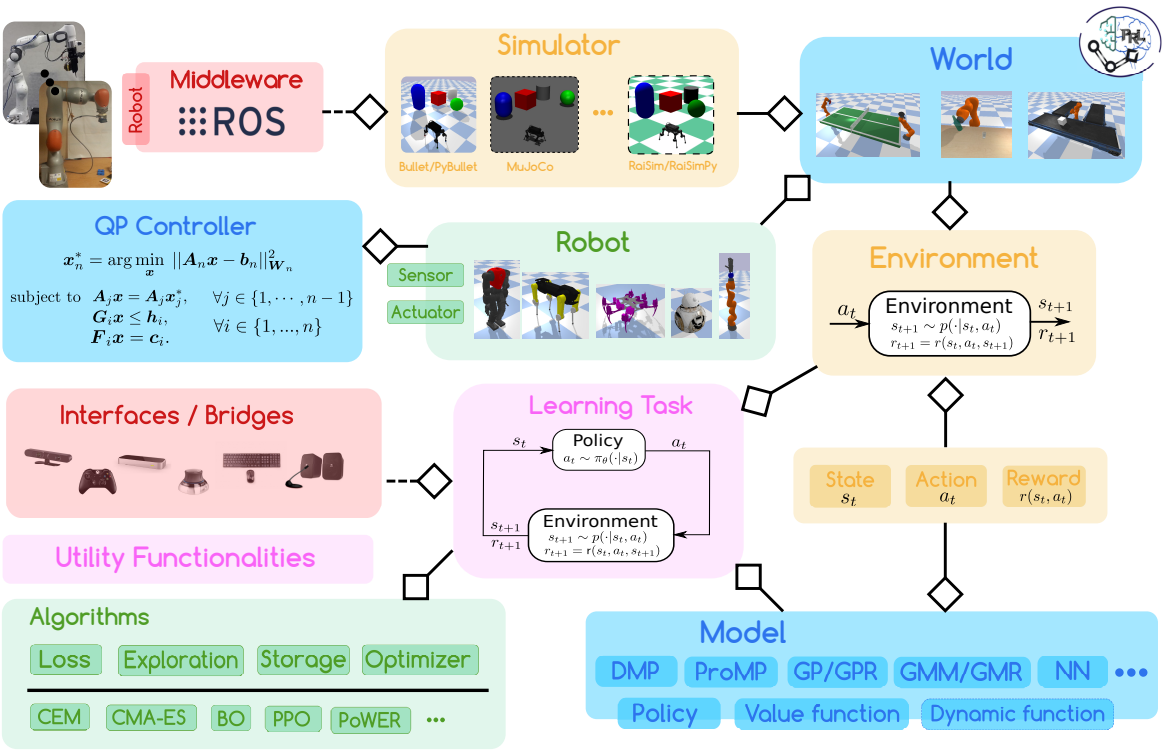

Figure 2.1 Overview of the *PyRoboLearn* architecture. Dashed bubbles are possible additions (see the integration of some simulators for instance). Diamonds represent the aggregation relationship between two modules (the same as the ones used in UML diagrams).

2.3.1 Simulators

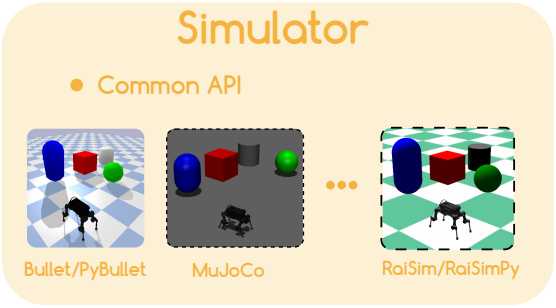

Figure 2.2 Middleware module.

The first axis is the specification of the simulator. Different simulators have been proposed: Gazebo [49] (with ROS [90]), V-REP/PyRep [95, 42], Webots [69], Bullet/PyBullet [17, 16], DART [55], RaiSim [39, 19], and MuJoCo [115] (the most popular). We chose to first work with PyBullet as it works in Python, and it is free and open-source. However, to avoid our code to be fully dependent on it, we provided an abstract interface that lies between the simulator and our framework such that any other simulators can inherit it, allowing for easy integration in the future (e.g., MuJoCo or RaiSim). Due to its popularity, gym [12] was also wrapped inside PRL, to make it suitable with our framework in RL scenarios.

The curious reader might wonder why not use one simulator and stick to it. First, simulators differ by their physics engine, the contact model (soft contacts versus hard contacts), as well as the solver used to solve the underlying optimization problem. Using the same controller but in different simulators can result in different behaviors being observed. Second, researchers around the world use sometimes different simulators than the one used by the user. This can be problematic when comparisons with other works have to be provided. This often require the user to learn multiple simulator APIs, and migrate the code of other researchers into the user own's code. To solve that problem, our framework abstracts all these simulator by providing a common API. While currently, PyBullet is the only simulator fully supported, integration of other popular simulators including MuJoCo, Dart, and Raisim[4] are ongoing and partial supports are provided for these.

We adopted an object-oriented programming (OOP) approach to implement the various modules and components in PRL. The UML diagram of the simulator module is provided in Fig. 2.3. An abstract `Simulator` interface from which all the simulators inherit from has been implemented, as well as few classes that inherit from it such as the `Bullet` simulator are provided. This interface allows to decouple the rest of the code in PRL with the simulator being used. The `Simulator` class can also interacts with the middleware module described in next section.

[4]The Python wrappers of the RaiSim simulator have also be implemented by myself, and have been adopted by the original authors of the Raisim simulator. The wrappers are released under the MIT license and are available on Github at the following link: https://github.com/robotlearn/raisimpy.

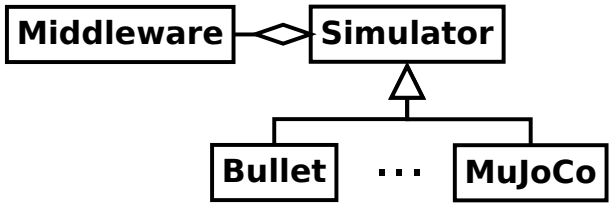

Figure 2.3 UML diagram for the Simulator module. Diamonds represents an *aggregation* relationship where a reference of an object is kept in the class pointed by the diamond, while the arrow represents an *inheritance* relationship, where a child class inherits the functionalities of a parent class, and has to implement the abstract methods.

Middleware

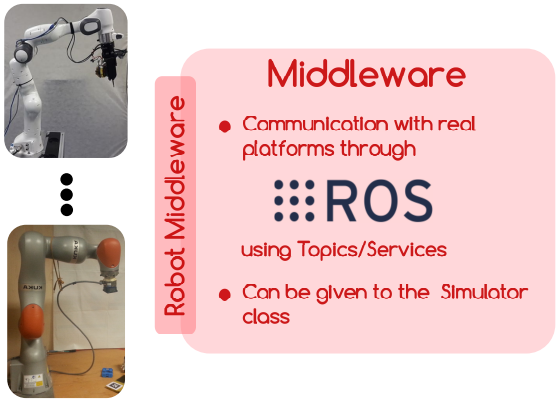

Figure 2.4 Middleware module.

A middleware module can be provided to the simulator which allows it to read from and write information on it. This module notably enables communication with real robotic platforms in an effortless and seamlessly manner. Popular middleware includes ROS [90] and YARP [27]. By providing the middleware to the simulator, it permits this last one to act as a bridge between real robotic platforms and the main control program.

In technical terms, an abstract `Middleware` interface providing the different method signatures used by the simulator has been implemented. A concrete middleware class only then needs to implement that abstract interface to be used with the rest of the framework. In PRL, we already implemented the ROS interface as this one is the most popular used middleware in the robotics community. To use a real robotic platform, each robot has to implement the `RobotMiddleware` interface which is used by the `Middleware` class. When the simulator require to have access to a certain robotic platform, it sent a request to the middleware. Upon receiving this request, the middleware then look if it has a reference to such instance of that `RobotMiddleware` class. If so, it then sends and/or receives information to/from the real robot through the middleware. The UML diagram representing the relationship between the `Middleware`, `Simulator`, and `RobotMiddleware` is provided in Fig. 2.5.

A concrete example on how to use the middleware and other modules on an imitation learning task is provided in Section 2.4.2.

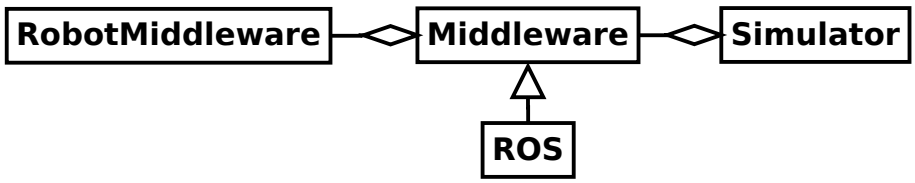

Figure 2.5 UML diagram for the middleware module. Note that the `Middleware` is optional and is mostly useful when the user wants to link the simulation with the real world.

2.3.2 Worlds

Figure 2.6 World module.

Once a simulator is provided, a world where robots and objects can interact is required. In our framework, only the world and robot instances interact with the simulator. The PyBullet simulator permits to load meshes in the world but did not provide any tool to generate terrains. This missing feature is important for robot locomotion tasks. We thus addressed this issue by providing tools to automatically generate height maps, which are subsequently used to produce meshes that are then loaded into the simulator.

The UML diagram describing the relationships between the `World` and other classes is provided in Fig. 2.7. The world will be provided to the learning environment (see Section 2.3.5).

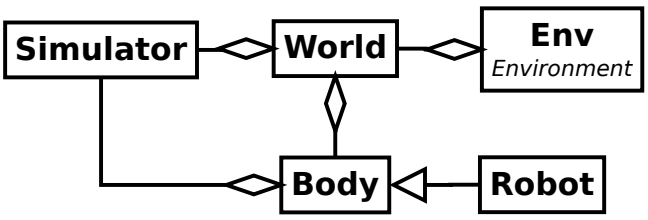

Figure 2.7 UML diagram for the world module. The `World` is composed of different bodies (including robots) and is a bridge to the `Simulator` class. The `World` will then be used by the learning environment to compute the next state of the simulator.

2.3.3 Robots

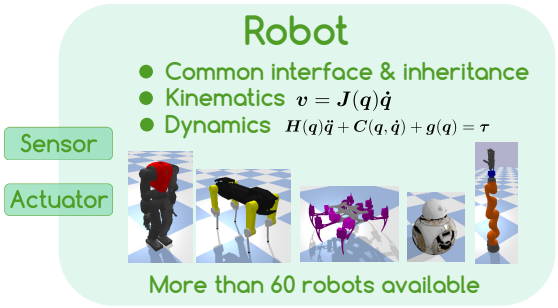

Figure 2.8 Robot module.

Robots are the active agents in our world, and more than 60 robots are provided in PRL. All inherit from a main robot class and are split into different categories: manipulators, legged robots, wheeled robots, UAVs, among others. Each of these categories is then divided into further subcategories. For instance, the legged robot class is inherited by classes representing biped, quadruped, and hexapod robots. Kinematic and dynamic functions allowing for motion and torque control are provided through the main interface. Notably, the user can access to the Jacobian matrix (see eq. 2.1) and other kinematic information such as the link's linear and angular velocities as well as the joint states. The user can also access to dynamic information including the inertia matrix, non-linear terms (such as the Coriolis and centrifugal forces, gravity, etc), and torques (see eq. 2.2).

$$\boldsymbol{v} = \boldsymbol{J}\dot{\boldsymbol{q}} \tag{2.1}$$

$$\boldsymbol{H}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \boldsymbol{c}(\boldsymbol{q},\dot{\boldsymbol{q}}) + \boldsymbol{g}(\boldsymbol{q}) = \boldsymbol{\tau} \tag{2.2}$$

Several sensors and actuators have also been implemented in the framework, including contact sensors, cameras, IMU, force/torque sensors, and others. For legged robots, we also provided functionalities that allows to compute and plot different indicators used in locomotion including the support polygon, center of mass (CoM) and its projection to the ground, zero-moment point (ZMP) [120], center of pressure (CoP) [89], foot rotation indicator (FRI) [33], centroidal moment pivot (CMP) [89], and others. We accessed online the URDF files of more than 60 robots, and implemented their corresponding classes through our framework (see Fig. 2.9). This unified structure of robotic platforms allows users to experiment rapidly with learning paradigms such as transfer learning.

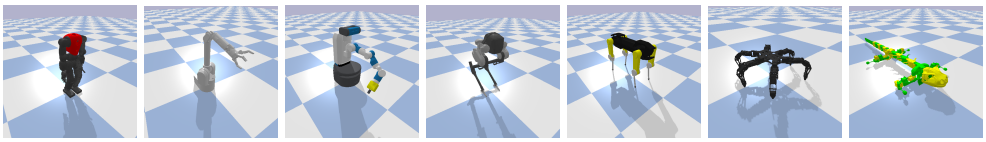

Figure 2.9 Seven of the 60+ available robots in PRL: manipulators, wheeled and legged robots.

The UML diagram representing the `Robot` class and its connection with the `Simulator` class is provided in Fig. 2.10. We use inheritance to represent the different types of robot in the framework. This permits to group robots by their types and provide the same functionalities.

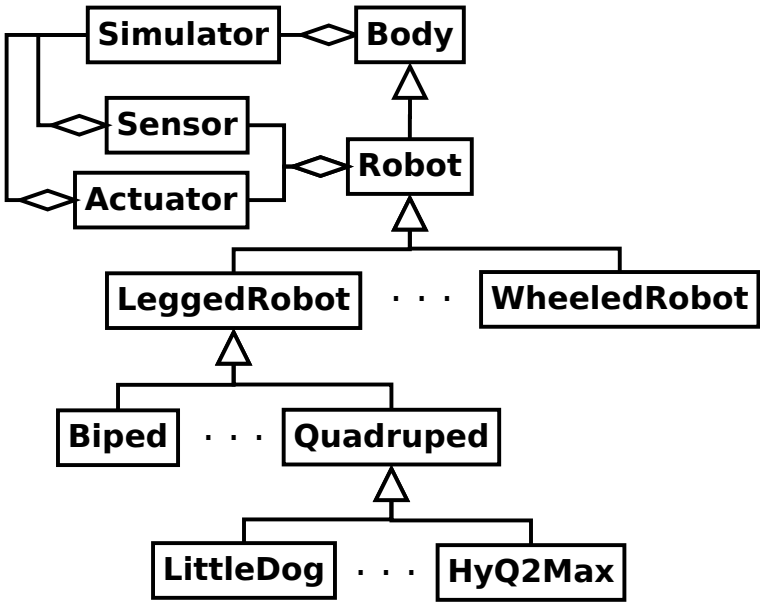

Figure 2.10 UML diagram of the `Robot` class and its link with the `Simulator` class. The `Robot` class accepts an instance of the `Simulator` class, and interacts with this last one to get kinematic, dynamic and sensory information from it, and send actuation values to it. The `Robot` class can possess some objects tha inherits from the `Sensor` and/or `Actuator` class. Robots are grouped by their types and inherits . Note that some robots might inherits from multiple parent classes. For instance, the Centauro robot [44] is a Centaur-like robot that has four legs (thus a quadruped) but also has a wheel attached to each leg's end-effector. Thus, in our framework, it inherits from both classes.

2.3.4 Quadratic Programming Control

Once a world and robots have been provided, we can already define some robotic tasks to solve. This requires the definition and use of controllers. A particular class of tasks include priority tasks which are represented as a constrained optimization problem, where the task consists to minimize a certain objective function while respecting several equality and inequality constraints. Most of the

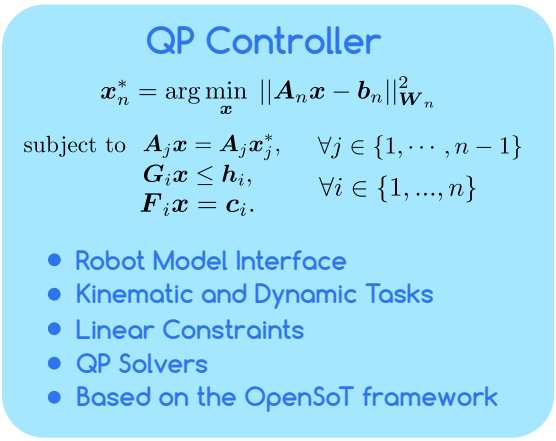

Figure 2.11 QP control module.

time, these are formulated as a quadratic
programming (QP) optimization problem,
which can be solved in real-time. Priority tasks are divided between kinematic and dynamic tasks, where the former only takes into account position and velocity information, while the latter also include dynamic information (such as forces and torques applied on the various bodies). The variables that are thus optimized by the optimization problem depends on the type of problem (kinematic or dynamic) we are dealing with. In the case of a kinematic task, the variables are often the joint (or end-effector) positions and/or velocities, while in the dynamic case, the variables are the joint accelerations and the (reaction) forces applied on the robot.

Formally, a quadratic program (QP) is presented in standard form as:

$$\begin{aligned} \boldsymbol{x}^* = \arg\min_{\boldsymbol{x}} ||\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}||^2_{\boldsymbol{W}} \\ \text{subject to} \quad \begin{matrix} \boldsymbol{G}\boldsymbol{x} \leq \boldsymbol{h} \\ \boldsymbol{F}\boldsymbol{x} = \boldsymbol{c} \end{matrix} \end{aligned} \tag{2.3}$$

where $\boldsymbol{x}$ is the vector being optimized (which can be joint positions, velocities, torques, or cartesian forces), $(\boldsymbol{A},\boldsymbol{b})$ are respectively the matrix projecting the variables to another vector space and a bias vector; these are usually defined by the system we are considering, and $\boldsymbol{W}$ is the positive semidefinite (PSD) symmetric weight matrix defined by the user. The tuples $(\boldsymbol{G},\boldsymbol{h})$ and $(\boldsymbol{F},\boldsymbol{c})$ respectively define inequality and equality constraints. Inequality constraints can include the lower bounds and upper bounds of $\boldsymbol{x}$ by setting $\boldsymbol{G}$ to be the identity matrix or minus this one, and $\boldsymbol{h}$ to be the upper or minus the lower bounds.

Priority tasks can be divided into two main categories:

- *Soft priority tasks*: each objective function is weighted by an importance weight where higher weights mean that we give more importance to the corresponding objective function. For instance, we might have a humanoid robot with two arms where each arm has to follow a specific trajectory and where we give the same importance to both tasks. With this type of tasks, the quadratic programming problem being minimized for n such tasks is given by:

$$\begin{aligned} \boldsymbol{x}^* = \arg\min_{\boldsymbol{x}} ||\boldsymbol{A_1}\boldsymbol{x} - \boldsymbol{b_1}||^2_{\boldsymbol{W_1}} + ||\boldsymbol{A_2}\boldsymbol{x} - \boldsymbol{b_2}||^2_{\boldsymbol{W_2}} + ... + ||\boldsymbol{A_n}\boldsymbol{x} - \boldsymbol{b_n}||^2_{\boldsymbol{W_n}} \\ \text{subject to} \quad \begin{matrix} \boldsymbol{G}\boldsymbol{x} \leq \boldsymbol{h} \\ \boldsymbol{F}\boldsymbol{x} = \boldsymbol{c} \end{matrix} \end{aligned}$$

Often, the PSD weight matrices $\boldsymbol{W_i}$ are just positive scalars w_i. This problem can be solved by stacking the $\boldsymbol{A_i}$ one of top of another as well as the $\boldsymbol{b_i}$ in the same manner, and organizing the weight matrices to be a square block diagonal matrix $\boldsymbol{W} = diag([\boldsymbol{W_1}, ..., \boldsymbol{W_n}])$, and solving $||\boldsymbol{Ax} - \boldsymbol{b}||^2_{\boldsymbol{W}}$. This is known as the augmented task. When the matrices $\boldsymbol{A_i}$ are Jacobians this is known as the augmented Jacobian.

- *Hard priority tasks*: the most important constrained optimization problem is first solved, then the next most important one is solved with an additional (optimization) constraint that the solution has to be in the solution space of the previous one. For instance, it is more important for a humanoid robot to maintain its balance than to follow perfectly a trajectory with its end-effector(s). This way of putting tasks on top of each other is known as the stack of tasks [63]. Hard priorities exploit the null-space of higher priority tasks. With this type of tasks, the QP problem for n tasks is defined in a sequential manner, where the first most important task will be first optimized, and then the subsequent tasks will be optimized one after the other. Thus, the first task to be optimized is given by:

$$\begin{aligned} \boldsymbol{x_1^*} = \arg\min_{\boldsymbol{x}} \; & ||\boldsymbol{A_1 x} - \boldsymbol{b_1}||^2 \\ \text{subject to} \quad & \boldsymbol{G_1 x} \leq \boldsymbol{h_1} \\ & \boldsymbol{F_1 x} = \boldsymbol{c_1} \end{aligned}$$

while the second next most important task that would be solved is given by:

$$\begin{aligned} \boldsymbol{x_2^*} = \arg\min_{\boldsymbol{x}} \; & ||\boldsymbol{A_2 x} - \boldsymbol{b_2}||^2 \\ \text{subject to} \quad & \boldsymbol{G_2 x} \leq \boldsymbol{h_2} \\ & \boldsymbol{F_2 x} = \boldsymbol{c_2} \\ & \boldsymbol{A_1 x} = \boldsymbol{A_1 x_1^*} \\ & \boldsymbol{G_1 x} \leq \boldsymbol{h_1} \\ & \boldsymbol{F_1 x} = \boldsymbol{c_1}, \end{aligned}$$

until the n most important task, given by:

$$\begin{aligned} \boldsymbol{x}_{\boldsymbol{n}}^{*} = \arg\min_{\boldsymbol{x}} \; & ||\boldsymbol{A}_{\boldsymbol{n}}\boldsymbol{x} - \boldsymbol{b}_{\boldsymbol{n}}||^2 \\ \text{subject to} \quad & \boldsymbol{A}_{\boldsymbol{1}}\boldsymbol{x} = \boldsymbol{A}_{\boldsymbol{1}}\boldsymbol{x}_{\boldsymbol{1}}^{*} \\ & \dots \\ & \boldsymbol{A}_{\boldsymbol{n-1}}\boldsymbol{x} = \boldsymbol{A}_{\boldsymbol{n-1}}\boldsymbol{x}_{\boldsymbol{n-1}}^{*} \\ & \boldsymbol{G}_{\boldsymbol{1}}\boldsymbol{x} \leq \boldsymbol{h}_{\boldsymbol{1}} \\ & \dots \\ & \boldsymbol{G}_{\boldsymbol{n}}\boldsymbol{x} \leq \boldsymbol{h}_{\boldsymbol{n}} \\ & \boldsymbol{F}_{\boldsymbol{1}}\boldsymbol{x} = \boldsymbol{c}_{\boldsymbol{1}} \\ & \dots \\ & \boldsymbol{F}_{\boldsymbol{n}}\boldsymbol{x} = \boldsymbol{c}_{\boldsymbol{n}}. \end{aligned}$$

By setting the previous $\boldsymbol{A}_{\boldsymbol{i-1}}\boldsymbol{x} = \boldsymbol{A}_{\boldsymbol{i-1}}\boldsymbol{x}_{\boldsymbol{i-1}}^{*}$ as equality constraints, the current solution $\boldsymbol{x}_{\boldsymbol{i}}^{*}$ won't change the optimality of all higher priority tasks.

Many control problems in robotics can be formulated as a quadratic programming problem. For instance, let's assume that we want to optimize the joint velocities $\dot{\boldsymbol{q}}$ given the end-effector's desired position and velocity in task space. We can define the quadratic problem as:

$$||\boldsymbol{J}(\boldsymbol{q})\dot{\boldsymbol{q}} - \boldsymbol{v}_{\boldsymbol{c}}||^2$$

where $\boldsymbol{v}_{\boldsymbol{c}} = \boldsymbol{K}_{\boldsymbol{p}}(\boldsymbol{x}_{\boldsymbol{d}} - \boldsymbol{x}) + \boldsymbol{K}_{\boldsymbol{d}}(\boldsymbol{v}_{\boldsymbol{d}} - \dot{\boldsymbol{x}})$ (using PD control), with $\boldsymbol{x}_{\boldsymbol{d}}$ and $\boldsymbol{x}$ being the desired and current end-effector's position respectively, and $\boldsymbol{v}_{\boldsymbol{d}}$ is the desired velocity. The solution to this optimization problem is the same solution given by inverse kinematics. By adding the extra term $||\dot{\boldsymbol{q}}||$[25], we obtain the damped least-squares inverse kinematics (DLS IK). Other tasks include cartesian CoM tracking, cartesian end-effector position tracking, postural positioning, and others.

Tasks can be further separated into 4 types, depending on the variables being optimized: velocity ($\boldsymbol{x} = \dot{\boldsymbol{q}}$), acceleration ($\boldsymbol{x} = \ddot{\boldsymbol{q}}$), torque ($\boldsymbol{x} = \boldsymbol{\tau}$), and cartesian force ($\boldsymbol{x} = \boldsymbol{f}$). Note that different type of tasks can sometimes be combined together; for instance, we can combine acceleration tasks with force tasks. This will create an optimization variable vector $\boldsymbol{x} = [\ddot{\boldsymbol{q}}^{\top}, \boldsymbol{f}^{\top}]^{\top}$ which can then be used with the joint space dynamic equation $\boldsymbol{\tau} = \boldsymbol{H}\ddot{\boldsymbol{q}} + \boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}})\dot{\boldsymbol{q}} + \boldsymbol{g}(\boldsymbol{q}) - \boldsymbol{J}^{\top}\boldsymbol{f}$ to get the equivalent joint torques to be applied on the robot.

[5]Note that $||\dot{\boldsymbol{q}}||^2$ can be rewritten as $||\boldsymbol{A}\dot{\boldsymbol{q}} - \boldsymbol{b}||^2$, where $\boldsymbol{A} = \boldsymbol{I}$ is the identity matrix and $\boldsymbol{b} = \boldsymbol{0}$ is the zero/null vector. This is equivalent to the objective function defined in eq. 2.3.4

The corresponding UML diagram is provided in Fig. 2.12. The design as well as several pieces of code have been heavily inspired by the OpenSoT framework [94, 70], however compared to their framework, it has completely been rewritten in Python with no dependencies on other frameworks/middlewares (such as the ADVR-superbuild [2], XBotCore [73], ROS [90], and YARP[27]). Our QP module is also completely free, open-source, heavily documented, and can easily be used with the other modules in our framework.

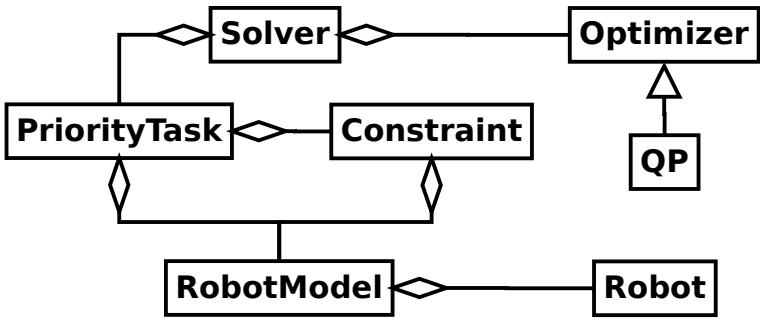

Figure 2.12 UML diagram of the priority tasks module. The `RobotModel` is an abstract interface that is used by the `PriorityTask` and `Constraint` classes to access to the various kinematic and dynamic information of the robot. An implementation of that interface which links the `Robot` class introduced in Section 2.3.3 has been implemented, enabling the use of the QP module with any robots in the PRL framework. The `PriorityTask` class represents the various QP objectives that can be used; this includes kinematic and dynamic tasks where the optimized variables can be the joint velocities, accelerations, torques or cartesian forces. The `Constraint` class implements the various equality and inequality constraints used in robotics, including for instance the joint limits. Tasks can be combined together using the methods or operators provided in the `PriorityTask` class. The operators are the same as the ones defined in the OpenSoT framework [94, 70]. Once the task or stack of task has been defined, it is given to the `Solver` class which uses an instance of the `Optimizer` interface (in our case the `QP` class) to solve the task. Note that the provided `Optimizers` are also used in other parts of the framework notably in the various learning algorithms, showing the benefits of adopting a modular approach.

An example using the QP module on an inverse kinematics problem is provided in Section 2.4.1. This example also shows the use case of the 3 previous modules, namely the simulator, world, and robots.

2.3.5 Learning Paradigms

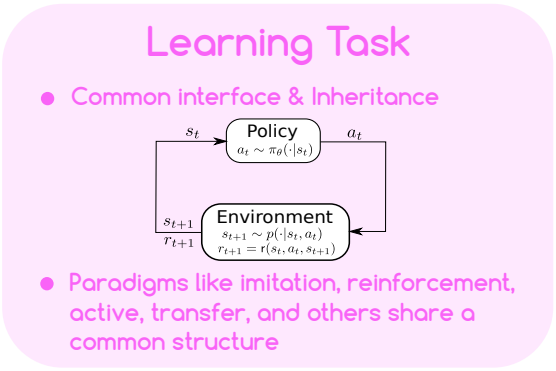

Figure 2.13 Learning task module.

Robot learning [87] is usually understood as the intersection of machine learning and robotics. This is divided into different learning paradigms according to different scenarios. The main categories are imitation learning (IL) and reinforcement learning (RL). IL [7] envisions a teacher demonstrating to an agent how to reproduce a task through few examples. RL [111, 18] conceives an agent that learns to perform a task by maximizing a total reward while interacting with its environment (see Fig. 2.14). Other paradigms include transfer learning [78, 114] (TL) where the knowledge acquired by an agent while solving a problem is transferred to solve a similar problem, and active learning [101] (AL) where an agent interacts with the user by querying new information about the task (e.g. demonstrations). While the foregoing approaches address different learning problems, they all share some common features (e.g. states, actions, policies and environments) which are conceptualized and abstracted in PRL. Similarly, their differences are introduced without loss of scalability through modules and composition. Additionally, different learning paradigms are evaluated using different metrics.

The RL paradigm is depicted in Fig. 2.14.

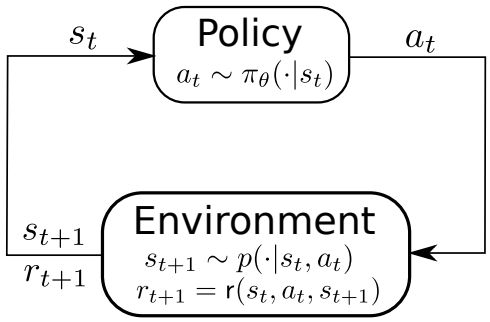

Figure 2.14 Policy and environment interaction in the RL paradigm. In the IL and AL paradigms, a reward function is not defined but a teacher is present to provide demonstration to the agent in the envorinment. While being different, these different paradigms share common features such as the states s_t, actions a_t, policy π, and environment.

States, Actions, Rewards

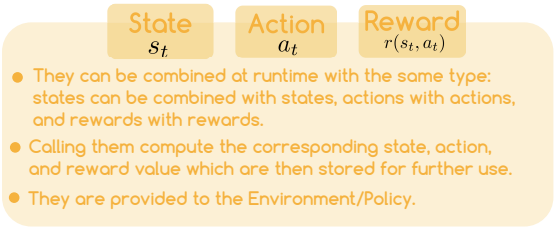

Figure 2.15 State, action, and reward modules.

States and actions are common components between the different learning paradigms. They are notably shared and used by the agent's policy and the environment (see Fig. 2.14). Reward functions on the other hand are mostly specific to the reinforcement learning or inverse reinforcement learning paradigm. Because they share these attributes, they have their own classes in our framework. Different states can be combined together at runtime. The same applies for actions and rewards providing a greater flexibility to users. It is for instance possible to define a state that account for the joint positions, velocities, and the base link position and orientation by just adding the states together at runtime. States and actions are among others given to the environment, policy, value function, and dynamic transition function.

Environments

The environment is one of the main concept in different learning paradigms such as imitation and reinforcement learning. As depicted on the right figure, the environment is notably responsible to perform a step in the world, compute the next state $p(s_{t+1}|s_t, a_t)$ given the agent's actions a_t, compute the rewards $r(s_t, a_t, s_{t+1})$ if provided, and others. A key component is the dynamic transition probability function $p(s_{t+1}|s_t, a_t)$ which is might not be accessible and is often unknown. If provided or learned, this is known as a model-based setting otherwise model-free. This distinction allows to categorize between model-based and model-free algorithms, notably in reinforcement learning.

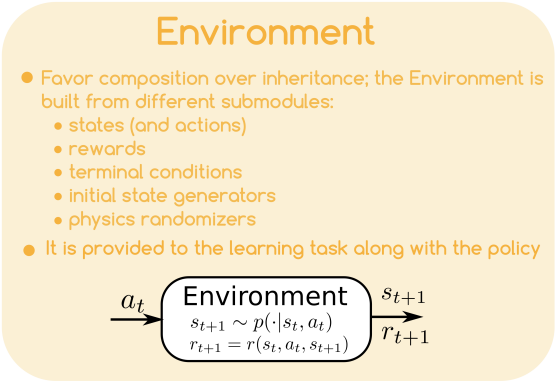

Figure 2.16 Environment module.

All environments inherit from the `Env` class which accepts as arguments at least the world, the state, and a possible reward function (if we are in the reinforcement learning case). These arguments can be provided at runtime making it easy to (re)use other modules, and render the framework very flexible (see our discussion on composition over inheritance). The corresponding UML diagram is provided in Fig. 2.17. Few other components that can be provided to the environment include a `StateGenerator` instance which generates states for the environment every time this one is resetted, a `PhysicsRandomizer` which enables

to randomize some physical properties of the simulation including link masses and center of mass positions, friction, and others, making it useful when transferring policies from simulation to reality, and a `TerminalCondition` which detects if an episode is over or not, and if it ended in a success or failure of the task.

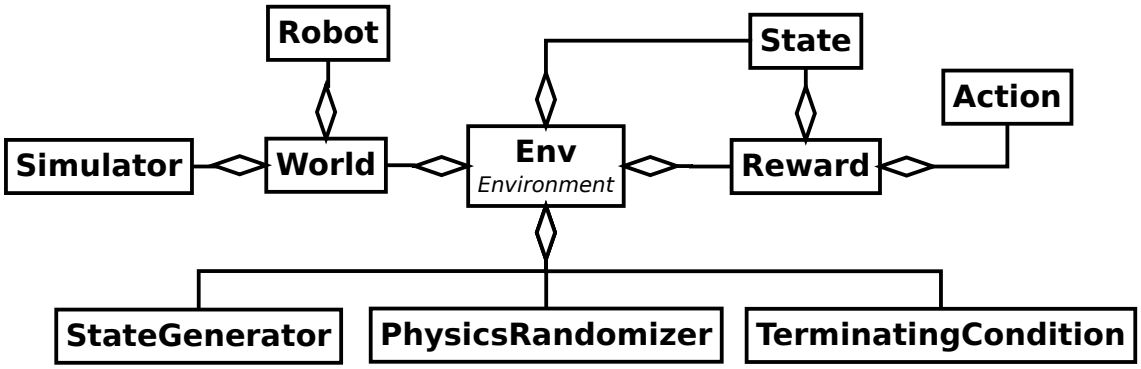

Figure 2.17 UML diagram of the `Environment` class and its link with other classes. This diagram highlights the modularity of our framework where small modules are built on top of others to build bigger modules. As it can be seen in this diagram, composition[6]is favored over composition. This is represented in the diagram by the diamonds instead of the arrows.

Learning Paradigms

To represent a learning paradigm, we describe it by an abstract `Task` class which encapsulates the previous defined environment, and the policy. A task inheriting from this class is then formulated for each paradigm as needed. The corresponding UML diagram is provided in Fig. 2.18.

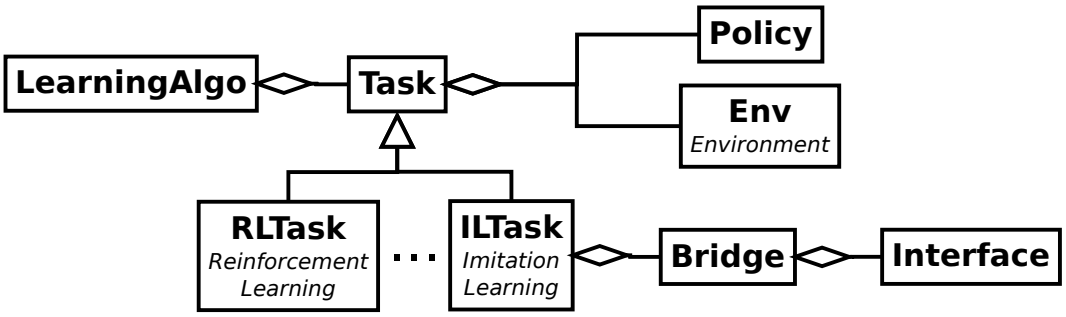

Figure 2.18 UML diagram of the learning `Task` (paradigm) class and its link with other classes.

[6]to be more specific, it is *aggregation*; a weaker version of the composition relationship. The difference between composition and aggregation boils down that in composition the object that composes another object is destroyed when the other object is destroyed. In aggregation, the object is instantiated outside and a reference is provided to the other object. Thus, in this case, destroying the other object does not affect the lifetime of the original object.

2.3.6 Interfaces and Bridges

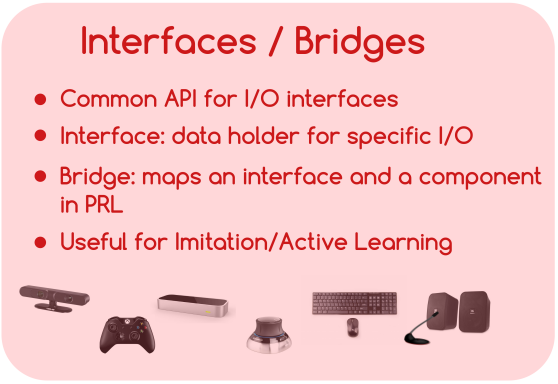

Figure 2.19 Interface module.

In IL, two predominant techniques are used to teach a robot a skill in order to perform a task: *teleoperation* and *kinesthetic teaching*. Teleoperation consists of commanding a robotic platform through a controller (from a remote location or in a virtual environment), while kinesthetic teaching considers a human guiding physically the robot (or a part of it) to perform the task. While the former is popular for its simplicity and use in simulation, it becomes difficult to use for robots with complex structures (such as humanoid robots). Recent advances in computer vision allow us to use cameras to control the robot, however the human-robot kinematic mapping remains a challenge. As for the latter, it has been hardly applied in simulations due to the lack of tools and haptic feedback.

In PRL, several *interfaces* have been implemented to enable the user to interact with the world and its objects, including robots. The implemented interfaces are resumed in Table 2.3. These tools are useful for different tasks and scenarios, especially in imitation and active learning. All the interfaces are completely independent of our framework and can be used in other applications. They act as containers for the collected data from the corresponding hardwares. *Bridges* connect an interface with a component, such as the world or an element in that world. For instance, a game controller interface permits to get data from the hardware, process it, and store it. The bridge can then map a specific controller event to a robot action. Moving a joystick up could mean to move a wheeled robot forward, or make a UAV robot ascend in the air. This separation of interfaces and bridges allows the user to only implement the necessary bridge without reimplementing the associated interface.

Interface	Instances
PC hardware	keyboard and mouse, SpaceMouse
audio/speech	speech recognition, synthesization, and translation
camera	webcam, asus-xtion, kinect, openpose
game controllers	Xbox, Playstation
sensors	Leap Motion, Myo Armband

Table 2.3 The various interfaces in PyRoboLearn

The UML diagram of the interfaces and bridges is provided in Fig. 2.20. Interfaces allows to receive or send the data from/to various I/O interfaces (such as mouse, keyboard, 3D space mouse, game controllers, webcam, depth cameras, sensors like LeapMotion, and

others). They all inherit from the abstract `Interface` class which has thread supports. If threads are not used, the user has to call the step method such that it reads the next value (i.e. these are not event-driven, i.e. the user controls when he/she want to get/set the data). Interfaces are independent from the other components in the PRL framework (with maybe at the exception of some utils methods), and as such can be used in other software. Bridges makes the connection between an interface and another component in PRL (like a robot or body in the world, or the world camera). Fundamentally, they accept as input an interface and the component, and the user details what should be done in that class. This allows to decouple the interface from the application part; e.g. the same game controller interface could be used to move a wheeled robot or quadcopter robot by providing two bridges (one for wheeled robots, and one for quadcopter robots). All the bridges inherit from the abstract `Bridge` class, and as with interface a `step` method can be called.

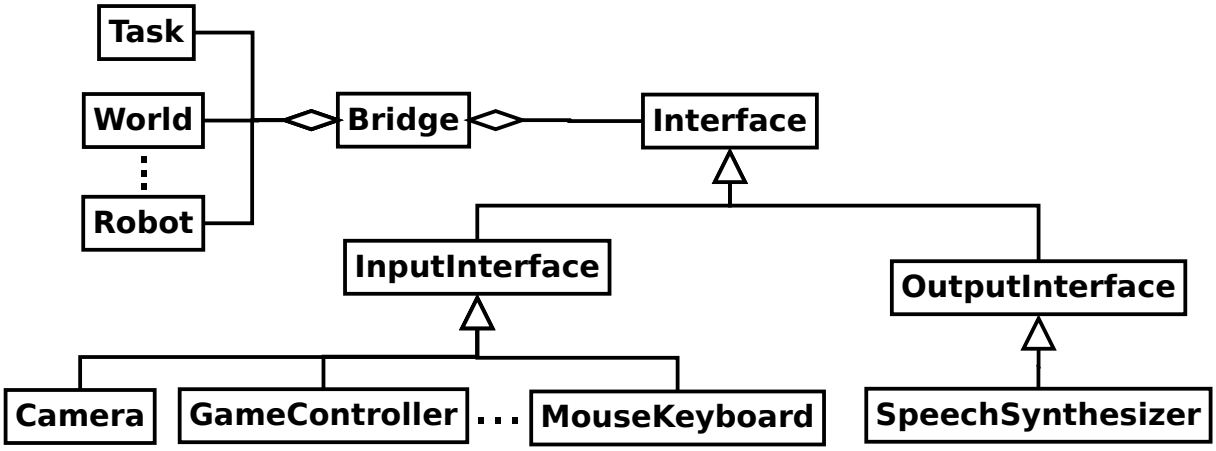

Figure 2.20 UML diagram of the `Interface` and `Bridge` classes and their link with other classes.

2.3.7 Learning Models

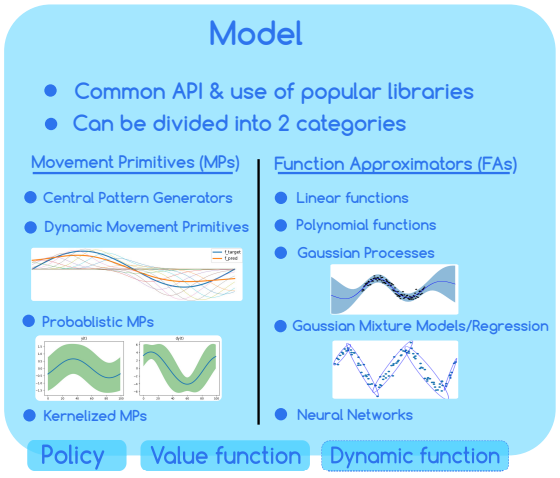

Figure 2.21 Model module.

We implemented several learning models in our framework through a modular approach. Learning models are characterized by (hyper-)parameters that are optimized through a training algorithm. All the implemented models are decoupled from PRL and can be used in other frameworks. To provide a better integration with the various modules in PRL, we build two abstraction

layers on top of the models. The first layer extends the models by receiving any created state and action module as inputs and/or outputs (in addition to normal Numpy arrays or Pytorch tensors). The second layer focuses on particular instances of these extended models, for example, a policy that receives as input the states and outputs the actions, or a state value-function approximator which receives a state as input and outputs a scalar value. In our framework the learning models are separated from the learning algorithms (see section 2.3) to avoid the models to be dependent on the training approach. We provide several learning models in our framework including movement primitives, e.g. central pattern generators [40], dynamic movement primitives [41], probabilistic movement primitives [80], kernelized movement primitives [38], and general function approximators such as linear and polynomial models, Gaussian processes [92, 31], Gaussian mixture regression [13], and deep neural networks [32, 84]. The various learning models available in our framework are resumed in Table 2.4. They can be subdivided into various categories based on their features. We notably compare if the models are parametric or non-parametric, linear with respect to the parameters, probabilistic or deterministic, generative or discriminative, step-based (i.e. general function approximator) or trajectory-based (i.e. movement primitive), if the parameters are interpretable, if they are universal approximators, and their data requirements in general.

The corresponding UML diagram is depicted in Fig. 2.22. The `Model` is the abstract class from which all models inherit from. The `Model` is provided to the `Approximator` class which couples the model with the `State` and `Action`. This `Approximator` is then used by the various components used in different learning paradigms, for instance, the `Policy`. When using movement primitives, the `Model` is directly provided to the corresponding `Policy` class without going through the `Approximator` class as movement primitives are not general function approximators (they are time-dependent function which often only accepts the phase or time as input).

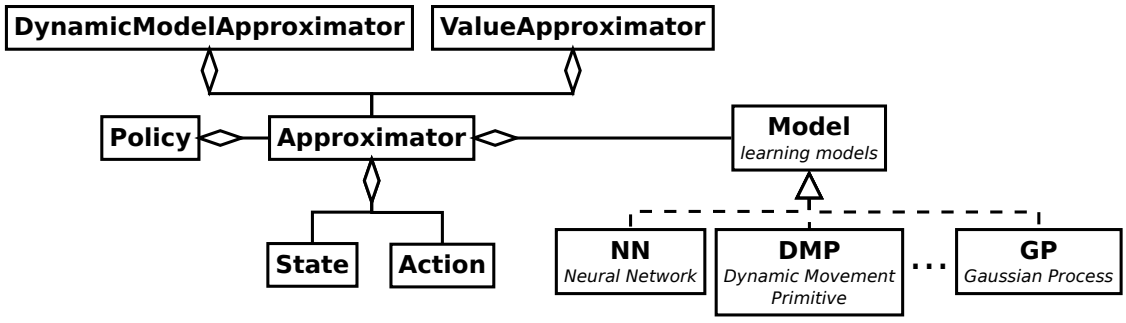

Figure 2.22 UML diagram of the `Model` class and its link with other classes such as the function `Approximator`, `Policy`, and other classes.

Models	parametric	linear	prob./det.	gen./disc.	step/traj.	interpretable	universality	data reqs
CPG [40]	✓	×	det.	disc.	traj.	✓/×	×	+
DMP [41]	✓	✓	det.	disc.	traj.	✓/×	×	+
ProMP [79, 80]	✓	✓	prob.	disc.	traj.	✓/×	×	++
KMP [38]	×	×	prob.	disc.	traj.	✓	✓	++
Linear models	✓	✓	det.	disc.	step	×	×	++
Polynomial models	✓	✓	det.	disc.	step	×	✓	+++
GMM/GMR [13, 14]	✓/×	✓/×	prob.	gen.	step	✓/×	✓	++
GMM/GMR [13, 14]	semi	✓/×	prob.	gen.	step	✓/×	✓	++
GP [92]	×	×	prob.	disc.	step	✓	✓	++
NN [32]	✓	×	det./prob.	disc./gen.	step	×	✓	+++++

Table 2.4 Comparison between different learning models based on different categories. We now explain what each column represents. The first column specifies if the model is parametric or non-parametric. Parametric models possess parameters that are optimized during training. Once trained the dataset can be discarded. This is not the case of non-parametric models which remembers the dataset or statistics computed on it (such as the mean and covariance). For these models, few hyperparameters are trained or provided. Usually, parametric models scale well with the number of samples while non-parametric performs extremely well with few samples. Some models such as GMM are semi-parametric; they have parameters (the priors in GMM) and also remember some statistics computed on the datasets (the Gaussians in GMM). The second column describes if the model is linear or not with respect to the parameters. Linear models are parametric models that are linear with respect to their parameters. This usually allows them to be learned efficiently using linear regression for instance. The third column specified if the learning models are deterministic or probabilistic. Deterministic models always return the same output given the same input, while probabilistic models return not only the predictions but the associated uncertainties as well. This is useful as it provides an estimate of how uncertain is the model about its prediction. The fourth column states if the models are generative or discriminative. Generative models allows to generate data by sampling them, while discriminative models do not. The next column check if we have step-based or trajectory-based models. Trajectory models are models that only accepts the phase or time as an input and generate the corresponding trajectory. Step-based model can accept other inputs as well. Interpretable models have parameters or hyperparameters that are interpretable. Universal models can approximate any function and are also known as general function approximators. The last column represents the number of data points usually required when training the corresponding model.

We now provide a brief description of each model available in the framework, and their possible use.

- *Linear models* are discriminative deterministic models given by $\boldsymbol{y} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$, where the weights $\boldsymbol{W}$ and bias term $\boldsymbol{b}$ are optimized. This is the simplest learning model, which can be used as a baseline when comparing other models.

- *Polynomial models* are a generalization of linear models (which has more expressive power), where the input space is projected into a polynomial space. This is given by $\boldsymbol{y} = \boldsymbol{W}\boldsymbol{\phi}(\boldsymbol{x}) + \boldsymbol{b}$. Note that the model is still linear with respect to the weights $\boldsymbol{W}$.

- *Principal component analysis* (PCA) is a non-parametric, deterministic, linear model which projects the data on a lower dimensional manifold such that it maximizes the projected variance. The principal components can be computed by applying singular value decomposition (SVD) on the data matrix.

- *Central pattern generators* (CPGs) [40] are movement primitives that produce rhythmic patterns, and are for instance used in robot locomotion. In our framework, they are modeled using the differential equations formulated in [40]. The CPG equations for a node i are given by:

$$\begin{aligned}
\dot{\phi}_i &= \omega_i + \sum_j a_j w_{ij} \sin(\phi_j - \phi_i - \varphi_{ij}) \\
\ddot{a}_i &= K_a(A_i - a_i) - D_a \dot{a}_i \\
\ddot{o}_i &= K_o(O_i - o_i) - D_o \dot{o}_i \\
\theta_i &= o_i + a_i \cos(\phi_i)
\end{aligned}$$

  where ϕ is the phase, ω is the desired angular velocity (desired frequency), A and a are the desired and current amplitude, O and o are the desired and current offset, K and D are the stiffness and damping gains (which are normally related such that the system is critically damped), w_{ij} and φ_{ij} are the coupling weights and phase biases, and finally, θ is the resulting (joint) angle (to be sent to the controller).

- *Dynamic movement primitives* (DMPs) [41] are a set of first and second order differential equations that model temporal-spatial trajectories. DMPs are composed of a canonical system which produce the phase s driving the transformation system, given by $\tau^2\ddot{y} = K(g-y) - D\tau\dot{y} + f(s)$, where τ is a scaling factor that allows to slow down or speed up the reproduced movement, K is the stiffness coefficient, D is the damping

coefficient, $y, \dot{y}, \ddot{y}$ are the position, velocity, and acceleration of a DoF, and $f(s)$ is the non-linear forcing term. The forcing term is often described as a weighted sum of basis functions $f(s) = \frac{\sum_i w_i \psi_i(s)}{\sum_j \psi_j(s)}$ from which the weights w_i are optimized to fit a certain trajectory.

- *Probabilistic movement primitives* (ProMPs) compared to DMPs encode in a probabilistic way movements [79, 80]. ProMPs are formulated by $\boldsymbol{y_t} = [q_t, \dot{q}_t]^\top = \boldsymbol{\Phi_t}^\top \boldsymbol{w} + \boldsymbol{\varepsilon_y}$, where $\boldsymbol{y_t} \in \mathbb{R}^{2\times 1}$ is the joint state vector at time step t, $\boldsymbol{\Phi_t} = [\boldsymbol{\phi_t}, \dot{\boldsymbol{\phi}}_\boldsymbol{t}] \in \mathbb{R}^{M\times 2}$ is the matrix containing the basis functions defined by the user and where M is the number of these basis functions, $\boldsymbol{w} \in \mathbb{R}^{M\times 1}$ is the weight vector on which we put a Gaussian prior distribution given by $\boldsymbol{w} \sim \mathcal{N}(\boldsymbol{\mu_w}, \boldsymbol{\Sigma_w})$, and $\boldsymbol{\varepsilon_y} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma_y})$ is the zero-mean Gaussian noise.

- *Gaussian mixture models* (GMMs) and *Gaussian mixture regression* (GMR) [13, 14] are semi-parametric, probabilistic and generative models. In robotics, they are often used to model trajectories by jointly encoding the time and state (position and velocity). They are mathematically formulated as $p(\boldsymbol{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\boldsymbol{\mu_k}, \boldsymbol{\Sigma_k})$ where K is the number of components, π_k are prior probabilities that sums to 1, $\mathcal{N}(\boldsymbol{\mu_k}, \boldsymbol{Sigma_k})$ is the multivariate Gaussian (aka Normal) distribution, with mean $\boldsymbol{\mu_k}$ and covariance $\boldsymbol{\Sigma_k}$. Learning is performed by maximizing the likelihood and finding the parameters $\boldsymbol{\theta} = \{\pi_k, \boldsymbol{\mu_k}, \boldsymbol{\Sigma_k}\}_{k=1}^{K}$. Gaussian mixture regression is obtained by conditioning the GMM over the input state.

- *Kernelized movement primitives* (KMPs) [38] are probabilistic and discriminative models that encodes a movement/trajectory using kernels. They are built by minimizing the Kullback-Leibler divergence between a parametric trajectory distribution $\mathcal{N}(\boldsymbol{\Theta}(\boldsymbol{x})^\top \boldsymbol{\mu_w}, \boldsymbol{\Theta}(\boldsymbol{x})^\top \boldsymbol{\Sigma_w} \boldsymbol{\Theta}(\boldsymbol{x}))$ (where the weights are normally distributed $\boldsymbol{w} \sim \mathcal{N}(\boldsymbol{\mu_w}, \boldsymbol{\Sigma_w})$) and a reference probability distribution $P_r(y|x)$ (which can be modeled using GMR), and using the kernel trick.

- *Gaussian processes* (GPs) [92] are non-parametric, probabilistic, and discriminative models that generalize the multivariate Gaussian distribution over finite dimensional vectors to an infinite dimensionality. This works by putting a prior distribution on the function $f \sim \mathcal{GP}(\mathbf{0}, \boldsymbol{K}(\boldsymbol{X}, \boldsymbol{X}))$ where K is the kernel matrix. These models are very useful when quantifying the uncertainty is required as well as when the data is scarce. This is notably the case in active learning.

- *Neural networks* (NNs) [32]. This includes different types of neural networks such as multilayer perceptrons (MLPs), convolutional neural networks (CNNs), recurrent neural networks (RNNs), autoencoders (AEs), and others. These parametric models have been very popular these recent years due to their expressive power, the abundance of data, and the recent computing power (due to GPUs) available to train these deep models. These models are universal approximators and can be formulated as $\hat{\boldsymbol{y}} = f_{NN}(\boldsymbol{x};\boldsymbol{W})$ where $\boldsymbol{x}$ is the input array, $\hat{\boldsymbol{y}}$ is the output array, and $\boldsymbol{W}$ are the weight and bias terms in the various layers being optimized.

2.3.8 Learning Algorithms

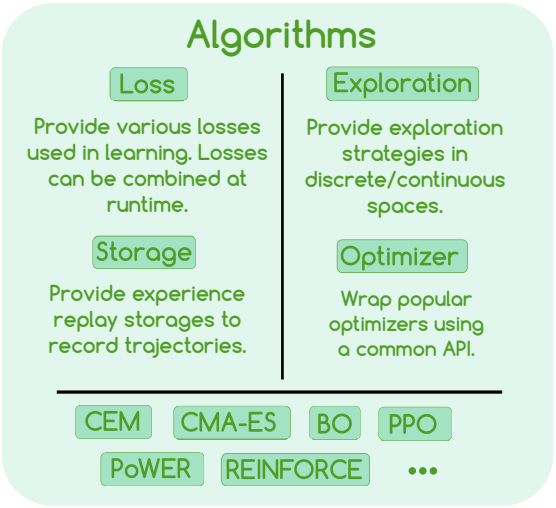

Figure 2.23 Algorithm module.

RL algorithms [111] depend on the structure of the environments/tasks, policies and models. Therefore, dependencies among them are unavoidable. We implement them in a modular way to stay consistent with PRL. To illustrate the modularity, let us consider the model-free PPO algorithm [100]. This algorithm has a lot in common with many other model-free on-policy algorithms but uses a different loss and action-space exploration strategy. This is often not exploited in current frameworks. In PRL, the loss can be redefined, any arithmethic operations can be performed on these loss instances, and provided at runtime to the PPO algorithm (through composition) without loss of generality. This results in faster experimentation to compare loss functions.

Because of the modular programming approach we undertook, we provide a different module for every concept, including the loss and exploration strategy. Moreover, as we favor composition over inheritance, we can parametrize the PPO algorithm with these modules, resulting in a more flexible framework that allows users to modify the algorithms and experiment with a wider range of combinations. The learning algorithms available in PRL include Bayesian optimization, evolutionary algorithms, model-free (on-/off-) policy search, among others.

For model-free reinforcement learning algorithms, we follow the taxonomy and structure of model-free policy search algorithms presented in [18]. Based on this survey, model-free policy search algorithms can be divided into 3 big phases: exploration, evaluation, and update of the policy (see Algo 1).

Algorithm 1 Model-free policy search algorithm

Input: Initial parameters $\boldsymbol{\theta}_0$

```
repeat
    Exploration: explore in the environment using the policy and
collect samples to learn from.
    Evaluation: evaluate the samples based on the estimator(s).
    Update: update parameters of the approximators (policy, value)
based on loss.
until θ_{i+1} ≈ θ_i
```

The associated UML diagram for model-free reinforcement learning algorithms is provided in Fig. 2.24. The 3 phases (exploration, evaluation, and update) presented in the taxonomy are represented as classes in our framework.

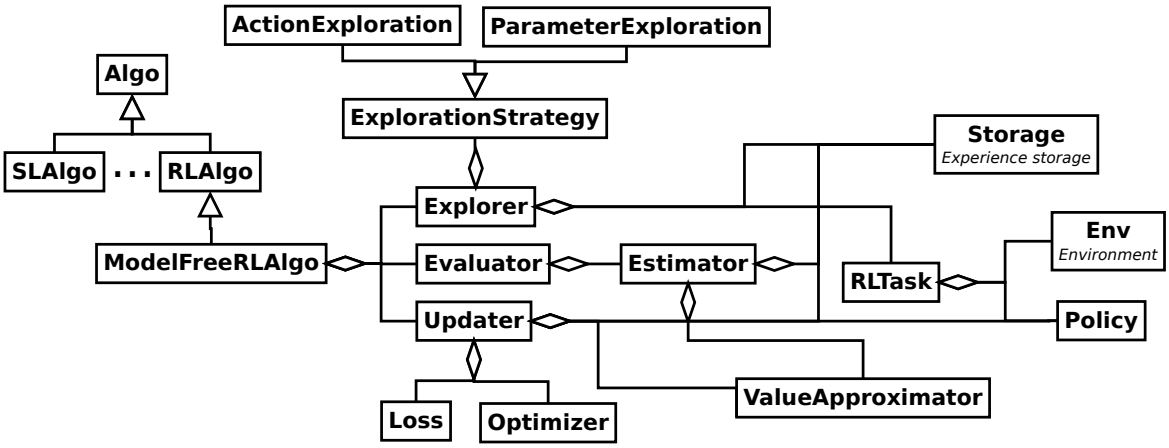

Figure 2.24 UML diagram of the `Algo` class and its link with other classes. In this diagram, we mostly focus on the model free reinforcement learning algorithms and show the many components that have been defined for these. In line with the taxonomy described in Algo. 1, we defined an `Explorer`, `Evaluator`, and `Updater` classes corresponding respectively to the 3 main phases in model free RL algorithms. Some of these components such as the `Loss` and `Optimizer` are re-used in other parts of the framework as well, demonstrating the benefits of undertaking a modular approach.

Because we use PyTorch as a backend, multiple other reinforcement learning libraries [36, 58] can also be used with our framework with some small extra effort.

2.3.9 Utility Functionalities

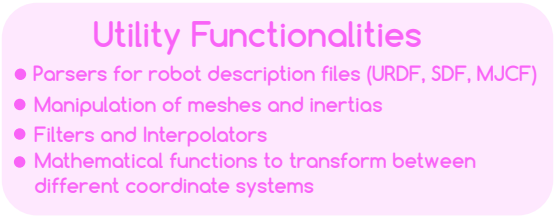

Figure 2.25 Utility module.

Finally, PyRoboLearn contains also different utility functions that for instance enables to parse different robot description files (such as URDFs, SDFs, MJCFs), performs different transformation between different frames, provides different orientation representation (rotation matrices, quaternions, roll-pitch-yaw, etc), defines data structures used in the framework, provides some plotting tools, and interpolation methods. These are completely independent from the framework and can be reused in other projects as well.

2.3.10 Framework Architecture

The whole UML diagram of the PyRoboLearn framework grouping all the previous mentioned modules and how they are linked is provided in Fig. 2.26. For better readability, not all the various implemented classes are reported on the diagram. PRL is currently released under the GPLv3 license, and has been tested on Ubuntu 16.04 and 18.04[7] with Python 2.7, 3.5, and 3.6[8]. The current release has around 100k lines of Python code. The link to the Github repository, documentation, examples, and videos are available through the main website https://robotlearn.github.io/pyrobolearn/.

[7] Parts of the framework have also been tested on Mac OSX (Mojave) and Windows 10, however this is in an experimental stage, and support for some input/output interfaces are currently not provided due to OS specific libraries.

[8] While the support for Python2.7 will end in 2020, some simulators such as Gazebo-ROS still have old libraries that are dependent on Python 2.7. The framework was designed to account for this problem.

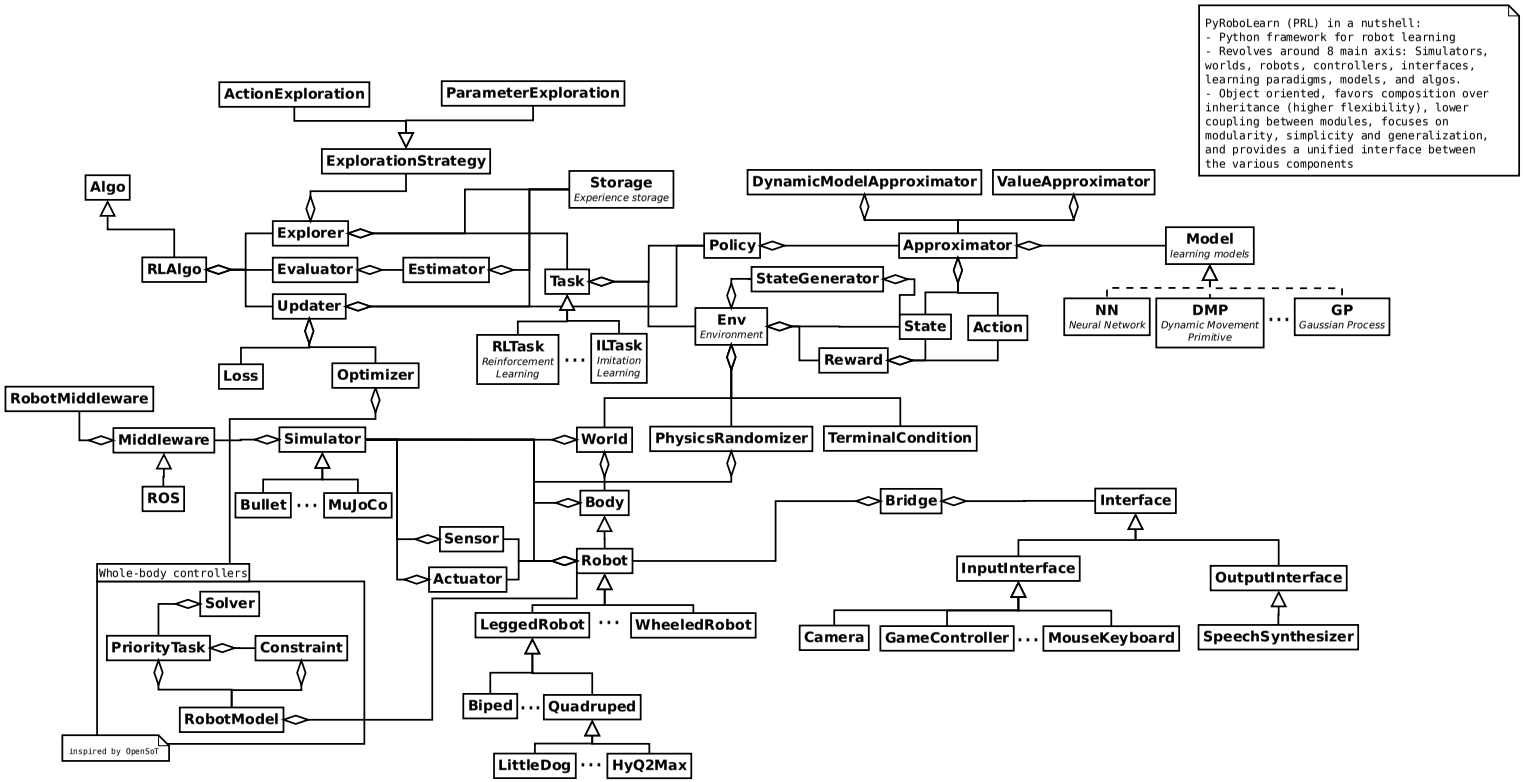

Figure 2.26 Current UML diagram of the *PyRoboLearn* framework. Some functionalities as well as classes which are less primordial are not reported here for a better readability of the diagram.

2.4 Experiments

In order to show some of the functionalities of our framework for robot learning, we demonstrate four use cases; a classical quadratic programming control task, an IL scenario, an RL task, and a scenario which combines these two last approaches to show the flexibility of our framework.

2.4.1 Quadratic Programming Control Task

In this first example, we show that we can also perform some robotic tasks with our framework without involving any learning. This is to show that the framework can also be used for non-learning tasks as well. In this example, we perform inverse kinematics on the end-effector of the Kuka robot where the goal is to follow a sphere moving in circle, using priority tasks. These are solved using the quadratic programming and the approach mentioned in Section 2.3.4. We define 4 different simple kinematic tasks, and show the obtained behavior by changing 2 lines of code (uncommenting one and commenting another one). The code is given in Listing 2.1, and the lines to be uncommented/commented are located between lines [30, 33].

We define the following tasks (objectives) to be optimized with respect to the joint velocities $\dot{\boldsymbol{q}}$:

1. the Cartesian task: $||\boldsymbol{J}(\boldsymbol{q})\dot{\boldsymbol{q}} - (K_p\boldsymbol{e} + \dot{\boldsymbol{x}}_{\boldsymbol{d}})||^2$, where $\boldsymbol{J}(\boldsymbol{q}) \in \mathbb{R}^{6\times N}$ is the Jacobian taken from the base to the distal link, K_p is the stiffness gain, $\boldsymbol{e} \in \mathbb{R}^6$ is the error which is the concatenation of the position error given by $\boldsymbol{e_p} = (\boldsymbol{x_d} - \boldsymbol{x})$ (with $\boldsymbol{x_d}$ being the desired position, and $\boldsymbol{x}$ the current position), and the orientation error given by (if expressed as quaternions $\boldsymbol{o} = \{s, \boldsymbol{v}\}$ where s is the real scalar part, and $\boldsymbol{v}$ is the vector part) $\boldsymbol{e_o} = s\boldsymbol{v_d} - s_d\boldsymbol{v} - \boldsymbol{v_d} \times \boldsymbol{v}$, and $\dot{\boldsymbol{x}}_{\boldsymbol{d}}$ is the desired cartesian velocity for the distal link with respect to the base link.

2. the Postural task: $||\dot{\boldsymbol{q}} - (K_p(\boldsymbol{q_d} - \boldsymbol{q}) + \dot{\boldsymbol{q}}_{\boldsymbol{d}})||^2$, where K_p is the stiffness gain and the subscript d means "desired".

3. the soft task which is a weighted sum of the 2 previously defined tasks, that is: $w_1||\boldsymbol{J}(\boldsymbol{q})\dot{\boldsymbol{q}} - (K_p\boldsymbol{e} + \dot{\boldsymbol{x}}_{\boldsymbol{d}})||^2 + w_2||\dot{\boldsymbol{q}} - (K_p(\boldsymbol{q_d} - \boldsymbol{q}) + \dot{\boldsymbol{q}}_{\boldsymbol{d}})||^2$.

4. the hard task composed of the cartesian and postural tasks, with the cartesian task having the highest priority, and the postural task having a least important priority.

We also provide the lower bounds and upper bounds to the joint velocities being optimized. This is expressed as the following inequality bound constraint: $\dot{\boldsymbol{q}}_{\boldsymbol{lb}} \leq \dot{\boldsymbol{q}} \leq \dot{\boldsymbol{q}}_{\boldsymbol{ub}}$, where $(\dot{q}_{lb}, \dot{q}_{ub})$ are the lower and upper bound on the joint velocities.

In the example below (see Listing 2.1), we do not take into account the orientation and the desired joint velocities for the tasks are set to 0. Therefore, the Cartesian task simplifies to $||\boldsymbol{J}(\boldsymbol{q})\dot{\boldsymbol{q}} - K_p(\boldsymbol{x}_{\boldsymbol{d}} - \boldsymbol{x})||^2$, and the Postural task becomes $||\dot{\boldsymbol{q}} - K_p(\boldsymbol{q}_{\boldsymbol{d}} - \boldsymbol{q})||^2$. Snapshots of the different tasks are provided in Fig. 2.27.

```python
import numpy as np
import pyrobolearn as prl

# create simulator
sim = prl.simulators.Bullet()

# create world
world = prl.worlds.BasicWorld(sim)

# create robot
robot = world.load_robot('kuka_iiwa')

# define useful variables for IK
link_id = robot.get_end_effector_ids(end_effector=0)
joint_ids = robot.joints  # actuated joint
wrt_link_id = None  # robot.get_link_ids('iiwa_link_1')
q_idx = robot.get_q_indices(joint_ids)

# create sphere to follow
x_des = np.array([0.5, 0., 1.])
quat_des = np.array([0., 0., 0., 1.])
sphere = world.load_visual_sphere(position=x_des, radius=0.05, color
    =(1, 0, 0, 0.5), return_body=True)

# create priority task
model = prl.priorities.models.RobotModelInterface(robot)
cartesian_task = prl.priorities.tasks.velocity.CartesianTask(model,
    distal_link=link_id, base_link=wrt_link_id, desired_position=x_des
    , kp_position=50.)

q_desired = [1.448e-03, 2.790e-01, -2.199e-03, -1.013, 5.948e-04,
    -1.293, 3.882e-04]
postural_task = prl.priorities.tasks.velocity.PosturalTask(model,
    q_desired=q_desired, kp=50.)
```

```
# task = cartesian_task
# task = postural_task
# task = 1 * cartesian_task + 0.5 * postural_task
task = cartesian_task / postural_task

# define constraint
dq_limits = 2 * np.ones(len(q_desired))
constraint = prl.priorities.constraints.velocity.
    JointVelocityLimitsConstraint(model, dq_lower_bound=-dq_limits,
    dq_upper_bound=dq_limits)
task << constraint

# define solver
solver = prl.priorities.solvers.QPTaskSolver(task=task)

# define amplitude and angular velocity when moving the sphere
w = 0.01
r = 0.2

# run simulation
for t in prl.count():
    # move sphere
    sphere.position = np.array([0.5, r * np.cos(w*t + np.pi/2), (1.-r
    ) + r * np.sin(w*t + np.pi/2)])

  # update task
    cartesian_task.desired_position = sphere.position
    task.update(update_model=True)

  # solve QP task and return best solution
    dq = solver.solve()

    # set joint positions
    q = robot.get_joint_positions()
    q = q[q_idx] + dq * sim.dt
    robot.set_joint_positions(q, joint_ids=joint_ids)

    # step in simulation
    world.step(sleep_dt=sim.dt)
```

Listing 2.1 Inverse kinematics with the Kuka robot using QP priority tasks where the goal is to follow a moving sphere.

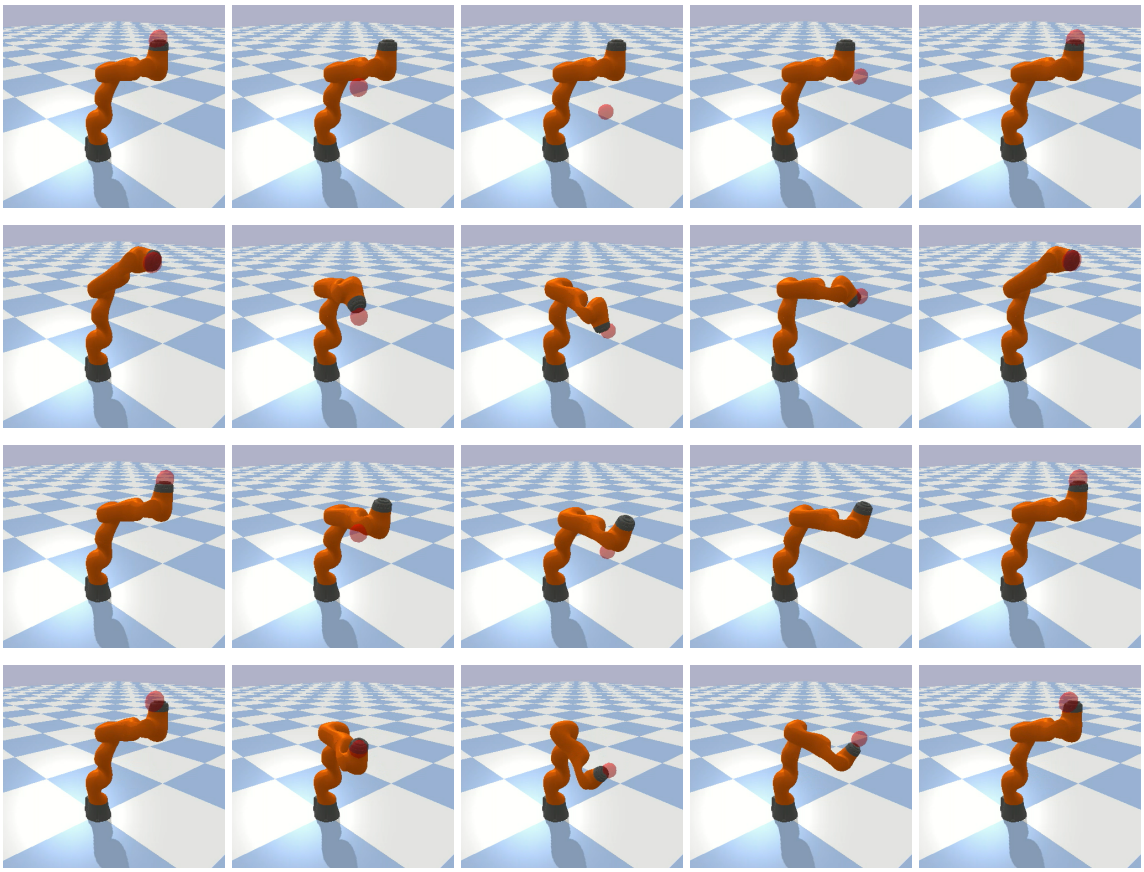

Figure 2.27 Snapshots of the previously defined priority tasks. From left to right, the first row shows the postural task at different time steps, the second row shows the cartesian task, the third row shows the soft task built using the cartesian and postural tasks previously defined and with weigths $w_1 = 1$ and $w_2 = 0.5$, and the fourth row shows the hard task built using these same cartesian and postural tasks. For the soft task, by setting different weights, different behaviors can be obtained.

2.4.2 Imitation Learning Task: Trajectory Tracking

The goal is to reproduce a demonstrated trajectory with IL using a *dynamic movement primitive* (DMP) model on a KUKA-LWR robot. The trajectories are demonstrated using the mouse interface (see Fig. 2.28). Both the training and reproduction phases can be watched on the PRL Youtube channel (see Section 2.6). The associated pseudo-code is given below in Algorithm 2.

Algorithm 2 illustrates the various building blocks and how they encapsulate each other. In this example, we first create an instance of the simulator, and then define a world in it. After this, a robot is loaded into the world. Next, we define the states and actions that are given to the policy and the environment. As we are in an IL setting, we need to collect and

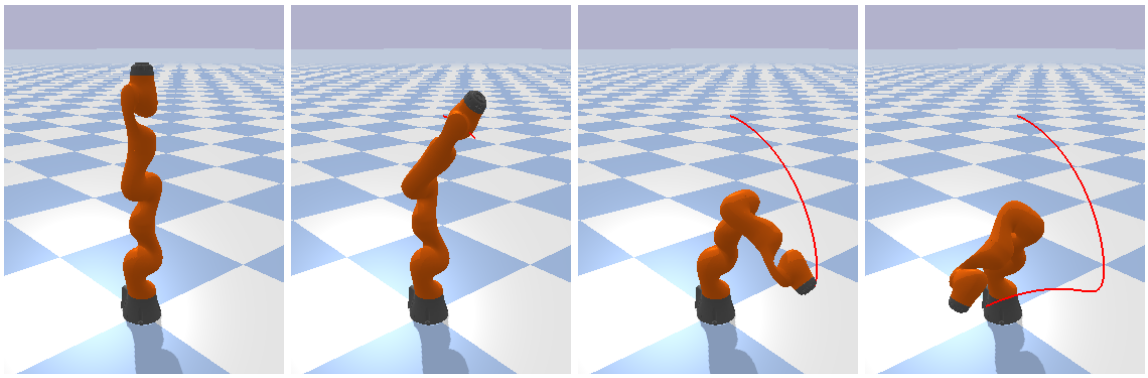

Figure 2.28 Reproduction of a trajectory learned from mouse-generated demonstrations using a DMP

Algorithm 2 Trajectory tracking with imitation learning

```
1: sim = Simulator()
2: world = BasicWorld(sim)
3: robot = world.load('robot_name_or_class')
4: state = PhaseState()
5: action = JointPositionAction(robot)
6: env = Env(world, state)
7: policy = Policy(state, action)
8: recorder = Recorder(state, action, rate)
9: interface = MouseKeyboardBridge(world)
10: task = ILTask(env, policy, interface, recorder)
11: task.record(signal_from_interface=True)
12: task.train()
13: task.test()
```

record the demonstrated data through the use of a recorder. We provide the trajectories using the mouse interface. Finally, an IL task can be fully defined with all the previous components. The step left is to train the policy using the demonstrated trajectory and reproduce the policy. The last three lines can be replaced by `task.run(signal_from_interface=True)` where the argument specifies that an event from the interface will send a signal to indicate when to record the data, train, and test the policy. Note that changes in the simulator, world, robot, state, action, policy, and/or interface would not affect the rest of the code due to the abstractions and modularity of our framework. This confirms the flexibility of PyRoboLearn.

A Concrete Example: Trajectory Tracking using the Middleware

To give a more concrete example of Algorithm 2, we provide the corresponding Python code, and also show how to use the middleware module to use a real robotic platform (in this case, the Franka Emika Panda robot). This is depicted in Listing 2.2. In this task, we teleoperate

the simulated robot by moving the real one through the ROS middleware, resulting in the simulated robot to move in accordance. The real sensed data are then collected in the simulator which is then used to train a DMP. Finally, the trained DMP is executed in the simulator leading the simulated and real robot to move. Snapshots of the demonstration and execution phases are provided in Fig. 2.29.

```
import pyrobolearn as prl

# variables
joint_ids = None  # None for all the actuated joints, or you can select which joint you want to move; e.g. [0, 1, 2]
num_basis = 20
rate = 30
use_real_robot = True

# create middleware
ros = prl.middlewares.ROS(subscribe=True, teleoperate=True)

# create simulator
sim = prl.simulators.Bullet(middleware=ros)

# create basic world (with gravity and floor)
world = prl.worlds.BasicWorld(sim)

# load Franka Emika Panda robot in the world
robot = prl.robots.Franka(sim)
world.load_robot(robot)
robot.print_info()

# create state/action
state = prl.states.ExponentialPhaseState(ticks=rate)
action = prl.actions.JointPositionAndVelocityAction(robot, joint_ids=joint_ids)

# create environment
env = prl.envs.Env(world, state)

# create DMP policy
policy = prl.policies.BioDiscreteDMPPolicy(action, state, num_basis=num_basis, rate=rate)

```

```
# create mouse-keyboard interface/bridge (used to start/stop the recording, and start the training)
interface = prl.interfaces.MouseKeyboardInterface(sim)
bridge = prl.bridges.BridgeMouseKeyboardImitationTask(world, interface=interface, verbose=True)

# create recorder
recording_state = prl.states.JointPositionState(robot, joint_ids=joint_ids) + prl.states.JointVelocityState(robot, joint_ids=joint_ids)
recorder = prl.recorders.StateRecorder(recording_state, rate=rate)

# create imitation learning task
task = prl.tasks.ILTask(env, policy, interface=bridge, recorders=recorder)

# record demonstrations in simulation/reality
task.record(signal_from_interface=True)
sim.disable_middleware()  # disable the middleware (get/set info only from/to simulation)

# train policy
task.train(signal_from_interface=False)

# plot what the DMP policy has learned by performing a rollout
policy.plot_rollout(nrows=3, ncols=3, suptitle='DMP position trajectories in joint space', titles=['q' + str(i) for i in range(robot.num_actuated_joints)], show=True)

# test policy in simulation
task.test(num_steps=rate*100, signal_from_interface=False)

# test policy on real robot
if use_real_robot:
    sim.enable_middleware()  # enable the real robot
    ros.switch_mode(subscribe=False, publish=True, teleoperate=True)
    task.test(num_steps=rate*100, signal_from_interface=False)
```

Listing 2.2 Imitation learning demonstration using DMPs and ROS with the Franka robot.

By providing the middleware to the simulator, it automatically tries to communicate with the real platform. Based on the given parameters to the middleware, it can then be used to